Desvendando o

CodeIgniter 4

Casa do Código

JONATHAN LAMIM

Sumário

- ISBN
- Prefácio
- O autor
- CodeIgniter 3: Produtividade na criação de aplicações web em PHP
- Amazon AWS: Descomplicando a computação na nuvem
- Marketing de Conteúdo: Estratégias para entregar o que seu público quer consumir
- Introdução
  - 1 Sobre o CodeIgniter
  - 2 Instalação e configurações iniciais
  - 3 Upgrade da versão 3.x para a 4.x
  - 4 Possíveis problemas na instalação
  - 5 Boas práticas de segurança
- Arquitetura
  - 6 Estrutura da aplicação
  - 7 Gerenciamento da aplicação
  - 8 Múltiplos ambientes
  - 9 Arquivos de configuração
  - 10 Modularização
  - 11 MVC
  - 12 Autoloading
  - 13 Serviços
  - 14 Requisições HTTP
  - 15 URLs
  - 16 Helpers
  - 17 Bibliotecas
  - 18 Logs
  - 19 Tratamento de erros
  - 20 Cache de páginas
- Bases da aplicação
  - 21 Controladores

    - 22 Filtros de controladores
    - 23 Rotas
    - 24 Views
    - 25 View Parser
    - 26 Models
    - 27 Entidades de Classe
- Banco de dados
    - 28 Configurações iniciais
    - 29 Conectando a aplicação com o banco de dados
    - 30 Operações básicas com banco de dados
    - 31 Query Builder
    - 32 Transactions
    - 33 Database Forge: Manipulando bancos de dados
    - 34 Migrations: Mantendo o banco de dados estruturado e organizado
- Estendendo o framework
    - 35 Estendendo classes do Core
    - 36 Estendendo o controlador
    - 37 Eventos
    - 38 Criando bibliotecas
    - 39 Criando helpers
    - 40 Trabalhando com arquivos de tradução
    - 41 Links úteis
    - 42 Conclusão

ISBN

Impresso e PDF: 978-65-86110-45-6

EPUB: 978-65-86110-48-7

MOBI: 978-65-86110-43-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio

Comecei a programar em PHP nos idos de 1999, ainda na versão 3. Na época, os scripts eram renomeados com a extensão .php3 no intuito de diferenciá-los das rotinas escritas em versões anteriores da linguagem.

Os sistemas eram grandes monólitos rodando sob um PHP geralmente compilado "na mão", pois eventualmente se dependia de extensões e bibliotecas externas, sendo que nem PECL existia.

Com o advento do PHP5 em 2004, a Orientação a Objetos melhorou absurdamente e começamos a rodar sites/sistemas com grande performance sob ZendEngine 2. Tal maturidade propiciou a comunidade projetos mais ambiciosos, principalmente no quesito reaproveitamento de código. Foi quando começaram, finalmente, a surgir os frameworks PHP - muito inspirados no Ruby on Rails, diga-se de passagem...

Das grandes vedetes da época, como CakePHP e Prado, o arcabouço que despontou foi ele: CodeIgniter, mantido pela extinta Ellis Lab. Há impressionantes 14 anos, o CodeIgniter já oferecia cerca de 20 bibliotecas, padronizando o envio de e-mails, propiciando validações, paginação de tela, mecanismo de template e XML-RPC.

Lembro que a documentação era muito bem organizada, de fácil navegação, o que ajudou enormemente na popularização do framework.

Hoje, em meio às grandes melhorias que o PHP nos trouxe (como Composer, xDebug, profiling, testes etc.), o CodeIgniter se apresenta com respaldo, reinventando-se, apoiado pela enorme comunidade e do ecossistema que o suporta.

O livro do amigo Jonathan Lamim é literatura obrigatória a todos os desenvolvedores que utilizam o CodeIgniter e também outros

frameworks, pois o livro demonstra o quão maduro ele se tornou (agora na versão 4.x) e os motivos pelos quais ainda é considerado um framework extremamente leve, seguro, completo e com baixa curva de aprendizado.

A comunidade CodeIgniter tem sorte de contar com este grande autor e profissional no processo de "evangelização" do CodeIgniter aos quatro ventos. A comunidade PHP brasileira cresce com tal publicação, pois a quantidade de livros sobre esta última versão é mundialmente escassa, nos prestigiando ainda mais.

Desejo aos colegas programadores que sorvam cada capítulo deste exclusivo material, muito bem elaborado e consistente.

Ari Stopassola Junior

Perito Forense Computacional

O autor

Figura -3.1: Jonathan Lamim

Jonathan Lamim é desenvolvedor de software desde 2005, e nesses 15 anos trabalhando com tecnologia teve a oportunidade de atender empresas tanto no Brasil como em países como Japão, Estados Unidos, Austrália, Canadá, Alemanha e Inglaterra.

Desenvolveu desde sites institucionais até sistemas de apoio para processos internos de instituições financeiras, e com isso pôde adquirir vasta experiência para compartilhar em seus 4 livros já publicados.

Também ajuda no desenvolvimento do CodeIgniter 4 e outros projetos open source, e produz conteúdo sobre tecnologia e desenvolvimento pessoal e profissional que publica em seu blog.

Além da experiência e do trabalho na área de tecnologia, Jonathan também possui formações em coaching e análise comportamental, conhecimentos que ele utiliza tanto no desenvolvimento de soluções de forma mais humanizada, quanto no trabalho como palestrante e trainer.

Para conhecer mais sobre o autor, siga-o nas redes sociais:

- **LinkedIn**: https://www.linkedin.com/in/jonathanlamim/
- **Instagram:** https://instagram.com/jonathanlamim
- **GitHub:** https://github.com/jlamim
- **Site pessoal:** https://jonathanlamim.com.br

Figura -3.2: Jonathan Lamim

Agradecimentos

Primeiramente, a Deus por me permitir aprender a cada dia mais através do meu trabalho e dos estudos e concluir mais este livro, que é uma das muitas formas de compartilhar o conhecimento adquirido e assim poder adquirir ainda mais, mantendo-me sempre em desenvolvimento.

Um agradecimento mais do que especial para minha esposa Juliana, que sempre me apoia em cada novo projeto e compreende

quando eu preciso passar mais tempo do que o normal trabalhando nos projetos pessoais e até mesmo nos projetos de clientes.

A toda a minha família pelo apoio dado em minha carreira e meus projetos, à comunidade brasileira de CodeIgniter que com suas dúvidas e busca constante por conteúdo me permite ajudá-los de forma direta e indireta e assim desenvolver os conhecimentos e conteúdo deste livro.

À equipe responsável pelo desenvolvimento do CodeIgniter 4, que me permitiu conhecer mais a fundo a nova versão e contribuir com o desenvolvimento da documentação. Assim pude adquirir ainda mais conhecimento e informação para compor este material.

Sou grato também a você que está lendo esta obra e está buscando aprender e se desenvolver cada vez mais, se não fosse por sua vontade de aprender este livro não existiria.

E para finalizar os agradecimentos, quero agradecer à Vivian Matsui por toda sua paciência, cuidado e profissionalismo com que atuou junto comigo na construção dessa obra. Sem a ajuda dela, este e nenhum dos outros 3 livros que publiquei pela Casa do Código teriam se tornado realidade. E ao amigo Ari Stopassola Junior, que escreveu o prefácio deste livro e é um grande divulgador do PHP e seus frameworks, meu carinho, gratidão e fraterno abraço.

CodeIgniter 3: Produtividade na criação de aplicações web em PHP

Figura -2.1: CodeIgniter 3: Produtividade na criação de aplicações web em PHP

O CodeIgniter 3.x é um framework MVC open source, escrito em PHP e mantido atualmente pelo British Columbia Institute of Technology e por uma grande comunidade de desenvolvedores ao redor do mundo. Com ele, é possível desenvolver sites, APIs e sistemas das mais diversas complexidades, tudo de forma otimizada, organizada e rápida. Suas bibliotecas nativas facilitam ainda mais o processo de desenvolvimento, e ainda permitem ser estendidas para que o funcionamento se adapte à necessidade de cada projeto.

Neste livro, você vai criar dois projetos completos, aplicando recursos, bibliotecas, vendo dicas de teoria e boas práticas com o framework CodeIgniter 3.x. Com os conhecimentos adquiridos aqui, você será capaz de desenvolver sites e sistemas com qualidade e rapidez.

Ficha técnica

Número de páginas: 350

ISBN: 978-85-5519-180-0

Disponível em formato digital e impresso.

https://www.casadocodigo.com.br/products/livro-code-igniter

Amazon AWS: Descomplicando a computação na nuvem

Figura -1.1: Amazon AWS: Descomplicando a computação na nuvem

Quando se trata de grandes aplicações, infraestrutura é um ponto muito importante, pois é preciso pensar em escalabilidade, gerenciamento e, principalmente, os serviços necessários para seu bom funcionamento. AWS ou Amazon Web Services é uma plataforma de serviços na nuvem que oferece soluções para armazenamento, redes e computação, em várias camadas. E o melhor de tudo, você pode administrar todos esses serviços através de uma interface web, ou também por APIs e linha de comando.

Neste livro, você vai aprender a configurar e integrar suas aplicações com os diferentes serviços da Amazon AWS, como o Amazon S3, AWS SDK, EC2, RDS, ElastiCache, Route 53, CloudFront, CloudWatch, Amazon SES e SNS. Este conjunto de ferramentas lhe possibilitará hospedar e gerenciar facilmente

aplicações dos mais variados tamanhos e com um custo possível de ser controlado.

Ficha técnica

Número de páginas: 234

ISBN: 978-85-5519-237-1

Disponível em formato digital e impresso.

https://www.casadocodigo.com.br/products/livro-amazon-aws

Marketing de Conteúdo: Estratégias para entregar o que seu público quer consumir

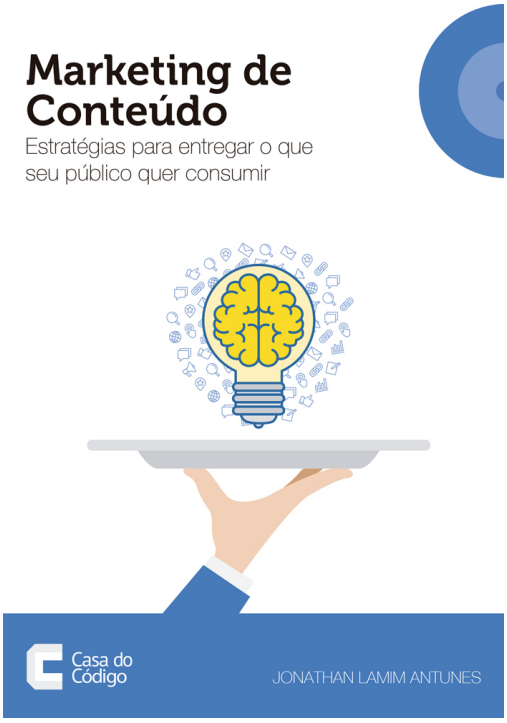

Marketing de Conteúdo
Estratégias para entregar o que seu público quer consumir
Casa do Código
JONATHAN LAMIM ANTUNES

Figura 0.1: Marketing de Conteúdo: Estratégias para entregar o que seu público quer consumir

Em um mundo no qual tudo é informação, saber entregá-la no melhor formato ao seu público é uma das chaves do sucesso para uma estratégia de marketing de conteúdo. Através dele, as marcas podem, além de conhecer melhor o seu público, entregar exatamente o que ele quer consumir, tanto no que diz respeito ao conteúdo como ao produto.

Neste livro, Jonathan Lamim compartilha seus conhecimentos sobre marketing de conteúdo para que você possa realizar um trabalho de qualidade e que gere resultados, seja como um consultor freelancer, atuando dentro de alguma empresa de marketing, ou no setor de marketing de empresas de outros ramos. Você verá como desenvolver técnicas como estudos de personas e fatores de

ranqueamento de SEO, junto das melhores estratégias para criar conteúdo de valor e medir os resultados de seu plano de marketing.

Ficha técnica

Número de páginas: 213

ISBN: 978-85-94188-36-6

Disponível em formato digital e impresso.

https://www.casadocodigo.com.br/products/livro-marketing-conteudo

Introdução

O CodeIgniter 4 não é apenas um upgrade da versão 3.x, ele é um novo CodeIgniter. Completamente reescrito em PHP 7, com nova estrutura, proporcionando mais performance e produtividade no processo de desenvolvimento das aplicações.

Nesta primeira parte você verá o conteúdo introdutório sobre o framework como requisitos técnicos, PSR Compliances, como instalar e fazer upgrade da versão 3.x para a 4.x entre outras informações importantes antes de colocar a mão na massa e começar a desenvolver utilizando a nova versão.

Capítulo 1
Sobre o CodeIgniter

O CodeIgniter é um framework para desenvolvimento de aplicações web utilizando a linguagem PHP. Ele tem como objetivo possibilitar aos desenvolvedores maior agilidade no processo de desenvolvimento através de um conjunto de bibliotecas nativas, compatibilidade com bibliotecas externas e uma estrutura de codificação propícia à produtividade e à alta performance na criação de aplicações web.

Uma das premissas deste framework é manter o processo de desenvolvimento flexível, podendo utilizar bibliotecas de terceiros, desenvolver suas próprias bibliotecas e estender - até mesmo substituir - partes da estrutura base *(core)* do próprio framework.

O CodeIgniter é ideal para você que:

- quer um framework mais enxuto;
- quer menos *dor de cabeça* com as configurações do framework para iniciar um projeto;
- quer uma estrutura de codificação menos restritiva;
- quer soluções simples e objetivas;

- quer programar se a necessidade de utilizar a linha de comando para tudo;
- quer evitar bibliotecas monolíticas, como o PEAR.

1.1 O que há de novo na versão 4.x

A versão 4.x do CodeIgniter está completamente reformulada, trazendo mais performance para as aplicações e para os processos de desenvolvimento. Além da nova estrutura de arquivos e codificação, o código do framework foi reescrito 100% em PHP 7, o que traz grandes melhorias de performance e até mesmo facilidades de codificação. Teremos um capítulo exclusivo no livro falando sobre a arquitetura do CodeIgniter 4.

Veja a seguir algumas imagens comparativas entre a versão 3.x e a versão 4.x.

Estrutura de diretórios

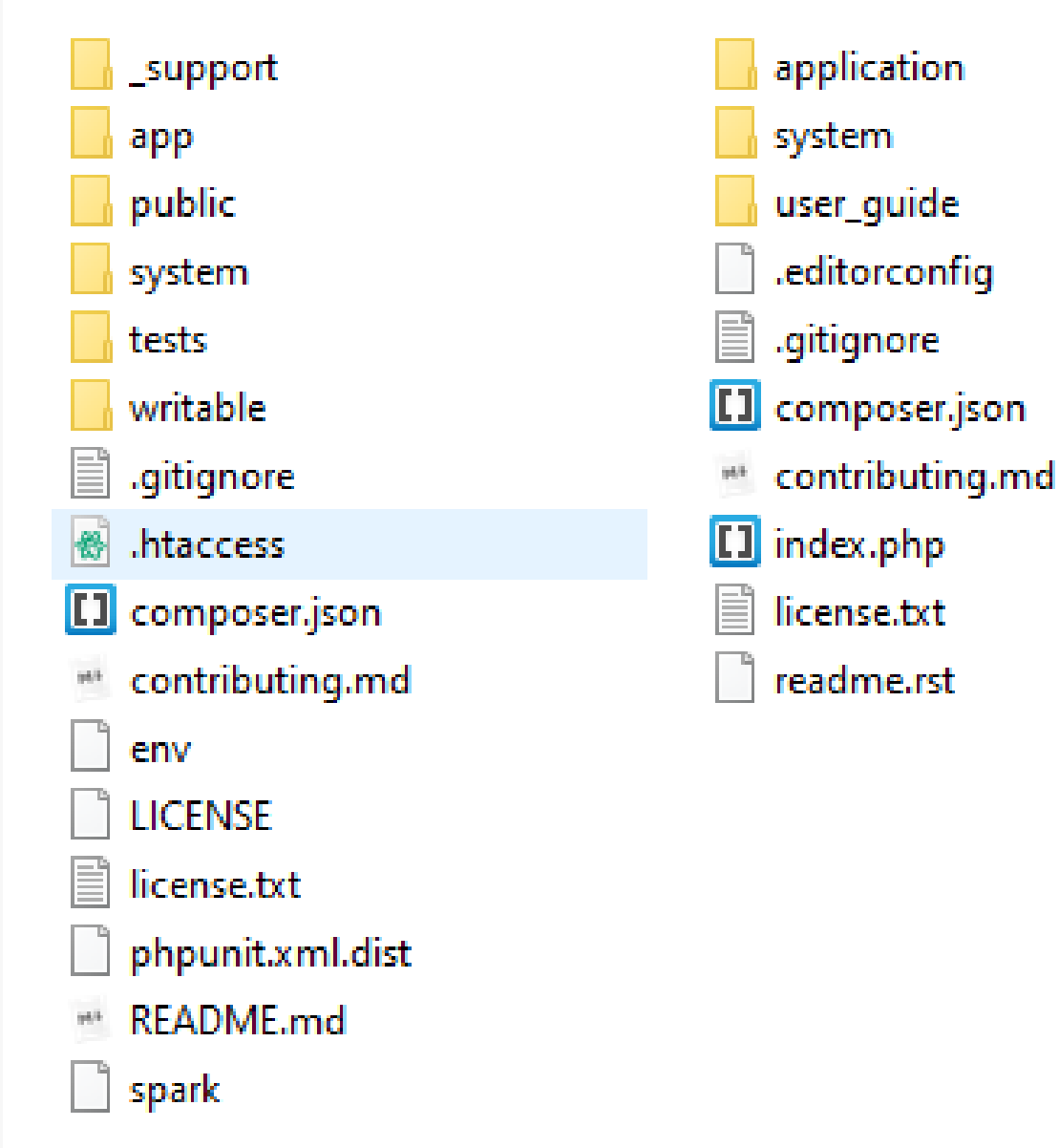
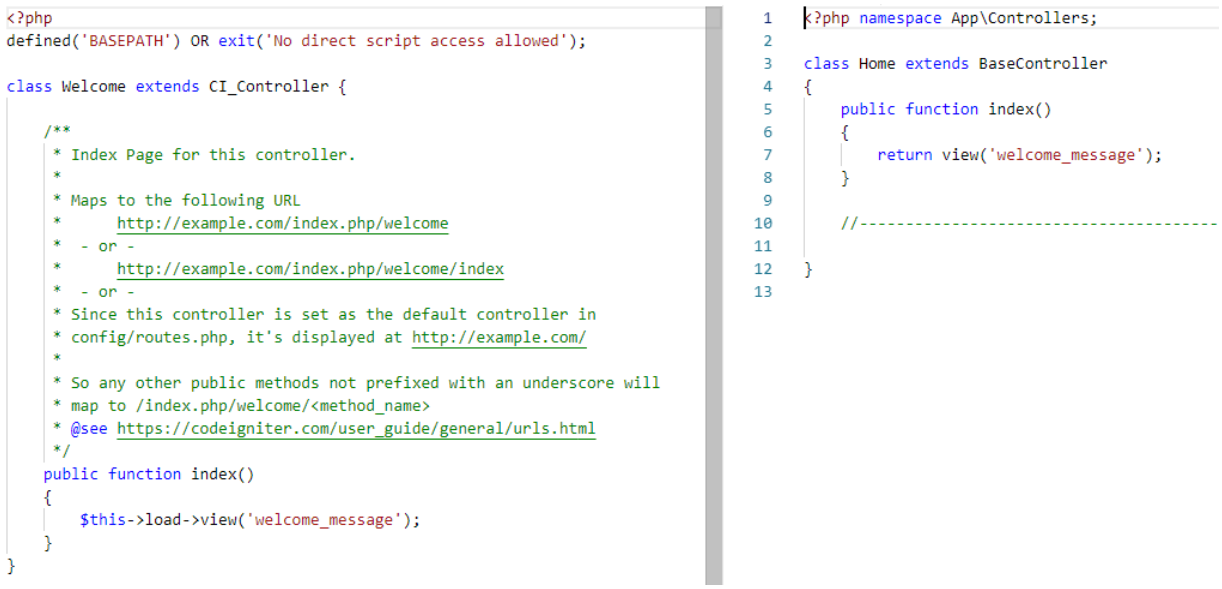


Figura 1.1: À esquerda, diretórios do CI 4 e à direita, estrutura de diretórios do CI 3.1.11

Código



```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');

class Welcome extends CI_Controller {

    /**
     * Index Page for this controller.
     *
     * Maps to the following URL
     *      http://example.com/index.php/welcome
     *  - or -
     *      http://example.com/index.php/welcome/index
     *  - or -
     * Since this controller is set as the default controller in
     * config/routes.php, it's displayed at http://example.com/
     *
     * So any other public methods not prefixed with an underscore will
     * map to /index.php/welcome/<method_name>
     * @see https://codeigniter.com/user_guide/general/urls.html
     */
    public function index()
    {
        $this->load->view('welcome_message');
    }
}
```

```php
<?php namespace App\Controllers;

class Home extends BaseController
{
    public function index()
    {
        return view('welcome_message');
    }

    //--------------------------------------

}
```

Figura 1.2: À esquerda, código do controlador Welcome após instalação do CI 3.1.11 e à direita, código do controlador Home após instalação do CI 4

1.2 Requisitos técnicos & PSR Compliance

Requisitos técnicos

Para utilizar o CodeIgniter 4 é necessário atender a alguns requisitos técnicos conforme especificado a seguir:

- PHP 7.2 ou versão mais recente;
- extensão `intl` instalada no servidor;
- estar com as extensões `php-json`, `php-mbstring`, `php-mysqlnd` e `php-xml` ativadas;
- caso pretenda utilizar a biblioteca `CURLRequest`, é necessário instalar a biblioteca `libcurl` do PHP.

Para bancos de dados, as recomendações são:

- MySQL (5.1+) via driver MySQLi;
- PostgreSQL através do driver Postgre;
- SQLite3 através do driver SQLite3.

PSR Compliance

Mesmo o CodeIgniter não sendo um membro da FIG (grupo criado para construir padrões para uso da linguagem PHP), o framework é compatível com várias propostas do PHP-FIG, também conhecidas como PSRs.

Veja a seguir com quais PSRs o CodeIgniter está em conformidade:

- PSR-1: Basic Coding Standard
- PSR-2: Coding Style Guide
- PSR-3: Logger Interface
- PSR-4: Autoloading Standard
- PSR-6: Caching Interface
- PSR-7: HTTP Message Interface

Capítulo 2
Instalação e configurações iniciais

Neste capítulo vamos tratar exclusivamente dos métodos de instalação do CodeIgniter 4 e de como fazer o upgrade da versão 3.x para a 4.x.

2.1 Instalação

Existem 2 maneiras de instalar o CodeIgniter 4: manual e via *Composer*. A instalação manual é para você que está em busca de um procedimento simples. Se você tem planos de utilizar pacotes de terceiros em seu projeto, a instalação via *Composer* é a mais recomendada.

Instalação manual

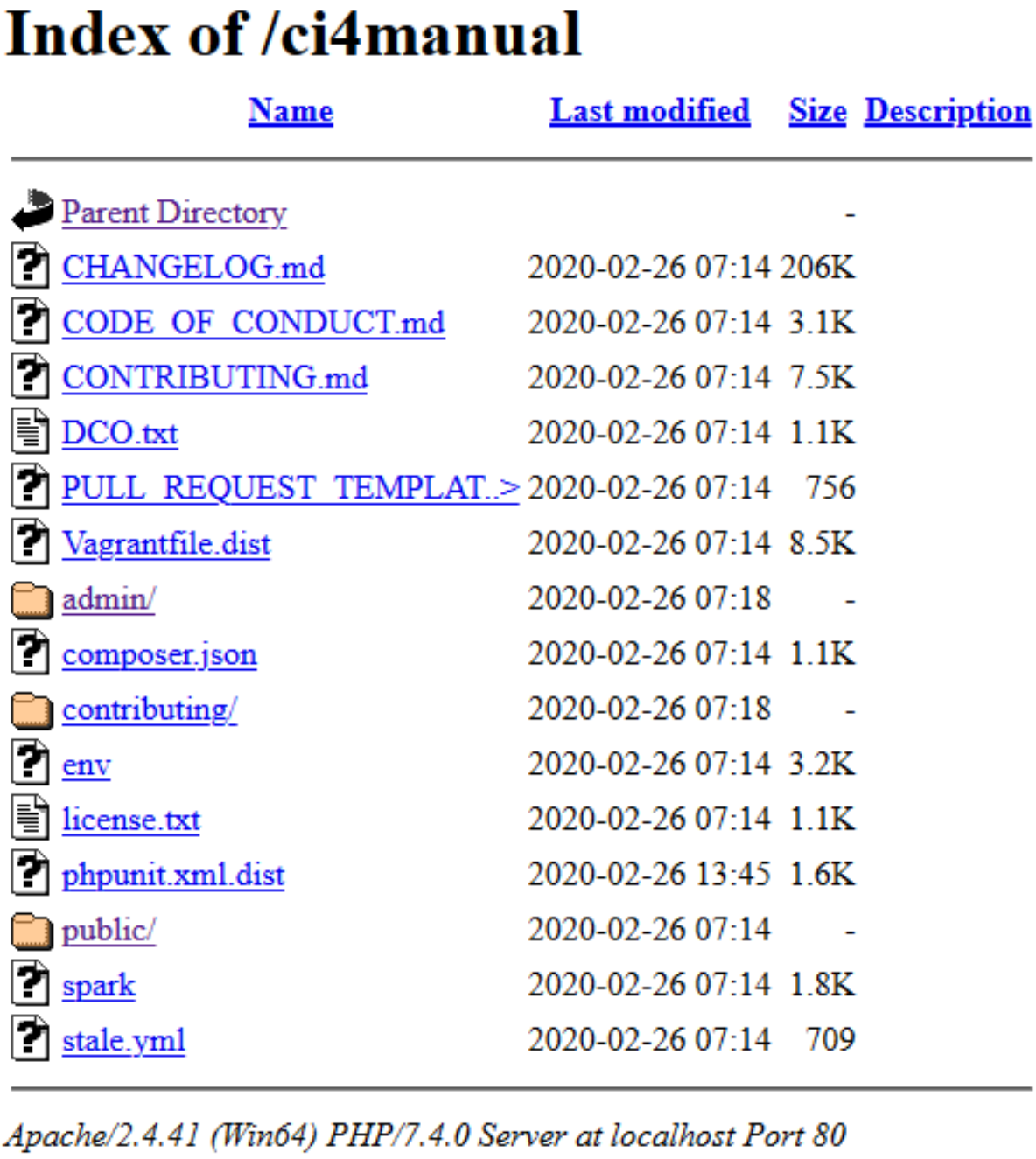
Para instalar de forma manual o CodeIgniter 4, você deverá realizar o download da versão mais recente dos arquivos no repositório oficial.

https://github.com/CodeIgniter4/framework/releases/latest

Após realizar o download, você deve extrair os arquivos para o diretório no servidor onde sua aplicação ficará instalada. Nomeie esse diretório como `ci4manual` para que você possa acompanhar mais facilmente os exemplos desse procedimento de instalação.

Diferente da versão 3.x, onde por padrão os arquivos públicos da aplicação ficavam na raiz do diretório, na versão 4.x os arquivos públicos ficam dentro do diretório `public`, enquanto todo o desenvolvimento da aplicação deve ser feito no diretório `app`.

Sendo assim, ao acessar *http://localhost/ci4manual* (ou a url do seu ambiente de desenvolvimento) você verá algo como na imagem a seguir:



Index of /ci4manual

Name	Last modified	Size	Description
Parent Directory		-	
CHANGELOG.md	2020-02-26 07:14	206K	
CODE_OF_CONDUCT.md	2020-02-26 07:14	3.1K	
CONTRIBUTING.md	2020-02-26 07:14	7.5K	
DCO.txt	2020-02-26 07:14	1.1K	
PULL_REQUEST_TEMPLAT..>	2020-02-26 07:14	756	
Vagrantfile.dist	2020-02-26 07:14	8.5K	
admin/	2020-02-26 07:18	-	
composer.json	2020-02-26 07:14	1.1K	
contributing/	2020-02-26 07:18	-	
env	2020-02-26 07:14	3.2K	
license.txt	2020-02-26 07:14	1.1K	
phpunit.xml.dist	2020-02-26 13:45	1.6K	
public/	2020-02-26 07:14	-	
spark	2020-02-26 07:14	1.8K	
stale.yml	2020-02-26 07:14	709	

Apache/2.4.41 (Win64) PHP/7.4.0 Server at localhost Port 80

Figura 2.1: Visualização no diretório raiz

Para visualizar a página inicial padrão do CodeIgniter 4 você deverá acessar *http://localhost/ci4manual/public* ou a url do seu ambiente

de desenvolvimento.

CodeIgniter

Home Docs Community Contribute

Welcome to CodeIgniter 4.0.2

The small framework with powerful features

About this page

The page you are looking at is being generated dynamically by CodeIgniter.

If you would like to edit this page you will find it located at:

```
app/Views/welcome_message.php
```

The corresponding controller for this page can be found at:

```
app/Controllers/Home.php
```

Go further

Learn

The User Guide contains an introduction, tutorial, a number of "how to" guides, and then reference documentation for the components that make up the framework. Check the User Guide !

Discuss

CodeIgniter is a community-developed open source project, with several venues for the community members to gather and exchange ideas. View all the threads on CodeIgniter's forum, or chat on Slack !

Contribute

CodeIgniter is a community driven project and accepts contributions of code and documentation from the community. Why not join us ?

Page rendered in 0.1462 seconds

Environment: production

© 2020 CodeIgniter Foundation. CodeIgniter is open source project released under the MIT open source licence.

Figura 2.2: Visualização no diretório public

Instalação via Composer

A instalação via *Composer* é feita através da linha de comando e você precisará ter o *Composer* instalado em sua máquina.

Para instalá-lo, acesse https://getcomposer.org/ e siga as instruções da documentação oficial.

Após ter instalado o *Composer*, abra o terminal onde você executa as linhas de comando no seu sistema operacional e acesse o diretório onde as aplicações ficam armazenadas em seu servidor. Estando dentro do diretório você executará o comando a seguir:

```
composer create-project codeigniter4/appstarter ci4composer
```

Esse comando criará o diretório `ci4composer` com toda a estrutura da aplicação, incluindo `phpunit` e todas as dependências dela que devem ser instaladas via *Composer*.

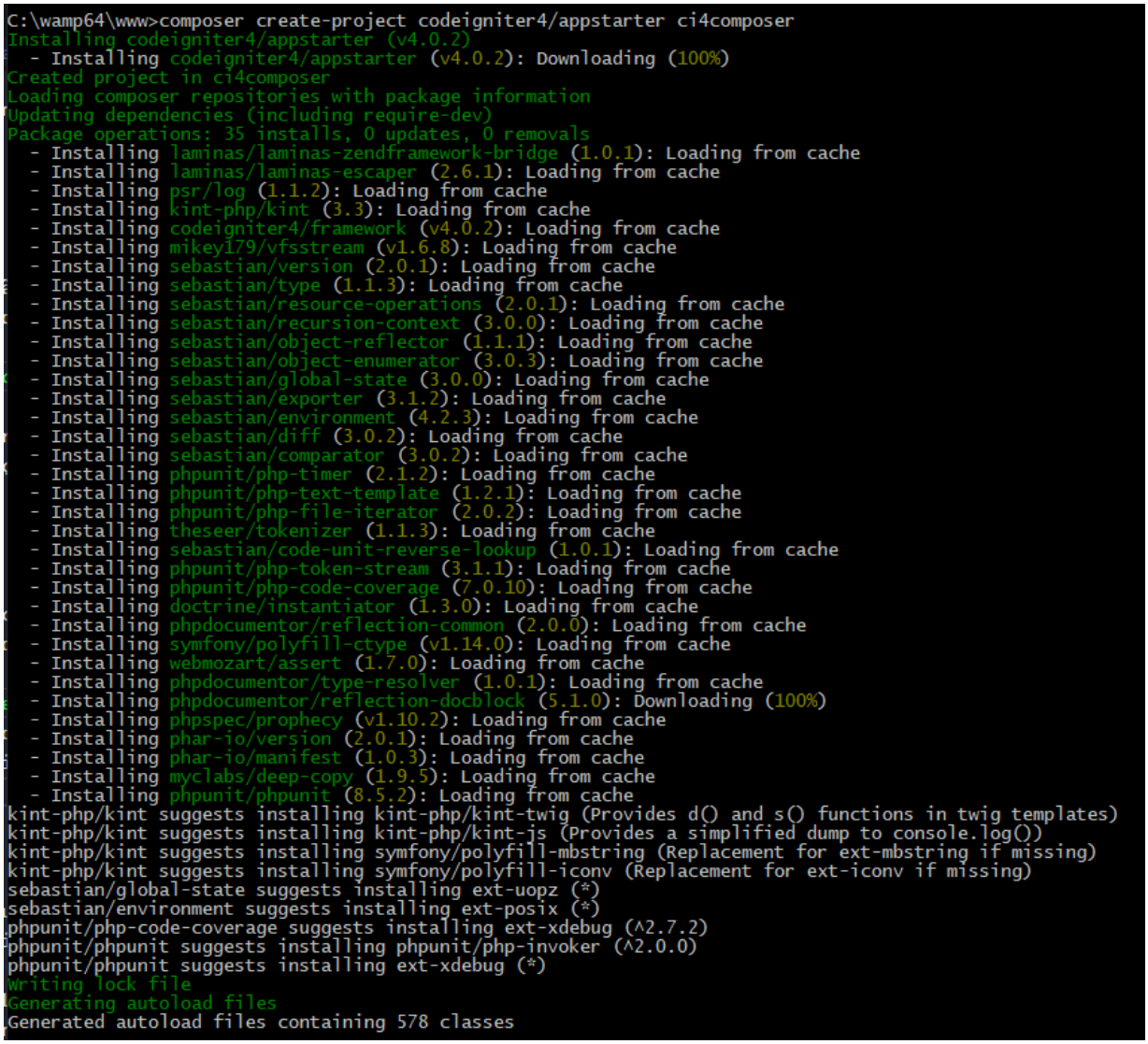

C:\wamp64\www>composer create-project codeigniter4/appstarter ci4composer
Installing codeigniter4/appstarter (v4.0.2)
- Installing codeigniter4/appstarter (v4.0.2): Downloading (100%)
Created project in ci4composer
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 35 installs, 0 updates, 0 removals
- Installing laminas/laminas-zendframework-bridge (1.0.1): Loading from cache
- Installing laminas/laminas-escaper (2.6.1): Loading from cache
- Installing psr/log (1.1.2): Loading from cache
- Installing kint-php/kint (3.3): Loading from cache
- Installing codeigniter4/framework (v4.0.2): Loading from cache
- Installing mikey179/vfsstream (v1.6.8): Loading from cache
- Installing sebastian/version (2.0.1): Loading from cache
- Installing sebastian/type (1.1.3): Loading from cache
- Installing sebastian/resource-operations (2.0.1): Loading from cache
- Installing sebastian/recursion-context (3.0.0): Loading from cache
- Installing sebastian/object-reflector (1.1.1): Loading from cache
- Installing sebastian/object-enumerator (3.0.3): Loading from cache
- Installing sebastian/global-state (3.0.0): Loading from cache
- Installing sebastian/exporter (3.1.2): Loading from cache
- Installing sebastian/environment (4.2.3): Loading from cache
- Installing sebastian/diff (3.0.2): Loading from cache
- Installing sebastian/comparator (3.0.2): Loading from cache
- Installing phpunit/php-timer (2.1.2): Loading from cache
- Installing phpunit/php-text-template (1.2.1): Loading from cache
- Installing phpunit/php-file-iterator (2.0.2): Loading from cache
- Installing theseer/tokenizer (1.1.3): Loading from cache
- Installing sebastian/code-unit-reverse-lookup (1.0.1): Loading from cache
- Installing phpunit/php-token-stream (3.1.1): Loading from cache
- Installing phpunit/php-code-coverage (7.0.10): Loading from cache
- Installing doctrine/instantiator (1.3.0): Loading from cache
- Installing phpdocumentor/reflection-common (2.0.0): Loading from cache
- Installing symfony/polyfill-ctype (v1.14.0): Loading from cache
- Installing webmozart/assert (1.7.0): Loading from cache
- Installing phpdocumentor/type-resolver (1.0.1): Loading from cache
- Installing phpdocumentor/reflection-docblock (5.1.0): Downloading (100%)
- Installing phpspec/prophecy (v1.10.2): Loading from cache
- Installing phar-io/version (2.0.1): Loading from cache
- Installing phar-io/manifest (1.0.3): Loading from cache
- Installing myclabs/deep-copy (1.9.5): Loading from cache
- Installing phpunit/phpunit (8.5.2): Loading from cache
kint-php/kint suggests installing kint-php/kint-twig (Provides d() and s() functions in twig templates)
kint-php/kint suggests installing kint-php/kint-js (Provides a simplified dump to console.log())
kint-php/kint suggests installing symfony/polyfill-mbstring (Replacement for ext-mbstring if missing)
kint-php/kint suggests installing symfony/polyfill-iconv (Replacement for ext-iconv if missing)
sebastian/global-state suggests installing ext-uopz (*)
sebastian/environment suggests installing ext-posix (*)
phpunit/php-code-coverage suggests installing ext-xdebug (^2.7.2)
phpunit/phpunit suggests installing phpunit/php-invoker (^2.0.0)
phpunit/phpunit suggests installing ext-xdebug (*)
Writing lock file
Generating autoload files
Generated autoload files containing 578 classes

Figura 2.3: Resultado na linha de comando após a instalação ser concluída

Caso você não precise do `phpunit` na sua aplicação, basta utilizar a mesma linha de comando inserindo `--no-dev` no final, como a seguir:

```
composer create-project codeigniter4/appstarter ci4composer --no-dev
```

Para atualizar a aplicação sempre que uma nova versão do CodeIgniter 4 for lançada, basta você acessar o diretório da aplicação `ci4composer` e executar o comando a seguir:

```
composer update
```

Se você criou o projeto utilizando `--no-dev`, recomendo que utilize também junto do comando `update`, assim você manterá as dependências atualizadas.

Além de poder criar o projeto usando a fonte `codeigniter4/appstarter`, você pode também utilizar `codeigniter4/framework` para fazer a instalação do CodeIgniter, a diferença entre elas é apenas a forma como os arquivos de sistema do framework são organizados, o que não interfere nem muda em nada o processo de desenvolvimento.

2.2 Configurações iniciais

Para iniciar os trabalhos de desenvolvimento da sua aplicação é preciso realizar algumas configurações iniciais, que explicarei a seguir.

Abra o arquivo `app/Config/App.php` e defina a URL base da sua aplicação. Seguindo os exemplos de instalação apresentados anteriormente (manual e via *Composer*), temos as seguintes URLs base:

- `http://localhost/ci4manual`
- `http://localhost/ci4composer`

```
<?php namespace Config;

use CodeIgniter\Config\BaseConfig;

class App extends BaseConfig
{
    public $baseURL = 'http://localhost/ci4manual';
}
```

Uma outra opção para configurar a URL base é através do arquivo `.env` que se encontra na raiz do projeto. Nele você deverá informar a URL base em `app.baseURL`.

```
#--------------------------------------------------------------------
# APP
#--------------------------------------------------------------------

app.baseURL = 'http://localhost/ci4manual'
```

Caso você pretenda utilizar banco de dados em sua aplicação (veremos com detalhes sobre esse assunto mais adiante) a configuração deve ser feita no arquivo `app/Config/Database.php` ou então no arquivo `.env` .

> **SOBRE O ARQUIVO .ENV** O arquivo `.env` tem como finalidade facilitar as configurações base da aplicação, de modo que nós, desenvolvedores, não precisemos ficar acessando a estrutura de arquivos dentro do diretório `app/Config` para ajustar configurações da aplicação.

Boa parte das configurações no CodeIgniter 4 continuam iguais às do CodeIgniter 3.x, então se você já possui alguma familiaridade com a versão 3.x vai ser muito tranquilo colocar sua aplicação para rodar.

No decorrer do livro, conforme formos vendo partes específicas, abordaremos as devidas configurações, mas por enquanto são essas que você precisa para colocar o framework para funcionar e começar a desenvolver.

2.3 Instalação de traduções das mensagens do sistema

Se você quer personalizar as mensagens do sistema para o idioma da sua aplicação, basta fazer o download da versão mais recente das traduções, descompactar o arquivo e copiar o conteúdo para o diretório `app/Language` .

Link para download:

https://github.com/codeigniter4/translations/releases

Feito isso, basta você informar qual o idioma da sua aplicação através dos arquivos de configuração `app/Config/App.php` alterando o valor para a variável `public $defaultLocale = 'en';` de acordo com o idioma desejado e devidamente instalado no diretório `app/Language`.

Capítulo 3
Upgrade da versão 3.x para a 4.x

O CodeIgniter 4 teve uma reescrita completa, envolvendo desde a codificação do framework até a sua estrutura de arquivos, o que o torna incompatível com versões anteriores. Ou seja, se você tem uma aplicação desenvolvida com a versão 3.x, simplesmente atualizar o core do framework para a 4.x não resolverá.

A versão 4.x, assim como a versão 3.x, mantém a filosofia "*lean, mean and simple*", e mesmo com as diferenças entre as versões o processo de upgrade é tranquilo de ser feito desde que a aplicação a ser atualizada tenha seguido os padrões do CodeIgniter no desenvolvimento.

A primeira coisa a ser feita é construir um novo ambiente para a aplicação a ser atualizada com a estrutura da versão 4.x devidamente instalada.

Após a criação da estrutura, você deve começar a mover os arquivos da aplicação a ser atualizada para dentro dos respectivos diretórios no novo ambiente, lembrando que a partir de agora o código da aplicação fica dentro do diretório `app`.

Como a versão 4.x foi criada para o PHP 7.2 e versões superiores, toda a estrutura é *namespaced*, exceto os *helpers*. Por isso, você precisará reescrever a estrutura de código para dar suporte aos `namespaces`.

Nesse processo de atualização, fique atento aos seguintes pontos:

Estrutura da aplicação

- A interpretação dos diretórios `app` (chamado de `application` na versão 3.x) e `system` continua da mesma forma;

- Foi inserido um novo diretório `public`, que é o diretório raiz da aplicação;
- Um diretório `writable` foi adicionado para armazenar dados como *cache*, *log* e *dados de sessão*;
- O diretório `application/core` foi removido de dentro do diretório `app`, uma vez que a forma de estender os componentes do framework foi alterada (veremos mais sobre extensão de componentes mais adiante).

Carregamento de classes

- Não existe mais um "superobjeto" que faz referência aos componentes da estrutura que faz a injeção destes nas classes;
- As classes são instanciadas sempre que necessário e os componentes são gerenciados via `Services`;
- O carregador de classes lida de forma automática com a localização das classes seguindo a PSR4 nos `namespaces` de nível superior `App` e `CodeIgniter`, desde que elas estejam nos diretórios corretos;
- Você pode configurar o carregamento das classes de acordo com a estrutura da sua aplicação, até mesmo usando o conceito de "HMVC".

Controladores

- Os controladores estendem `\CodeIgniter\Controller` e não mais `CI_Controller`;
- Eles não utilizam mais um construtor, a menos que isso faça parte de um controlador básico que você venha a criar;
- Na versão 4 você tem objetos de `Request` e `Response` para trabalhar (melhor do que na versão 3.x);
- Se você precisa de um controlador básico (`MY_Controller` na versão 3.x) você pode, por exemplo, criar um `BaseController` que estende de `Controller` e a partir daí estender os demais controladores de `BaseController`.

Models

- Os models estendem de `\CodeIgniter\Model` e não mais de `CI_Model`;
- O model na versão 4 possui mais funcionalidades, incluindo conexão automática ao banco de dados, CRUD básico, validação no próprio model e paginação automática de registros;
- A versão 4 também possui a classe `Entity` que você pode utilizar para um mapeamento de dados mais robusto das tabelas;
- Em vez de utilizar `$this->load->model('Nome_Model')`, como era a versão 3.x, agora você utiliza `$this->nome_model = new NomeModel()` seguindo o `namespace` do componente criado.

Views

- A forma de chamar uma view mudou de `$this->load->view('nome_da_view')` para `echo view('nome_da_view')`;
- Na versão 4 é possível utilizar views para exibir pequenos blocos de código;
- O *template parser* foi aprimorado.

Bibliotecas

- Em vez de utilizar `$this->load->library('nome_da_biblioteca')` como era a versão 3.x, agora você utiliza `$this->nome_da_biblioteca = new NomeDaBiblioteca()` seguindo o `namespace` do componente criado.

Helpers

- Os *helpers* continuam praticamente como na versão 3, porém alguns foram simplificados.

Estendendo a estrutura

- Não é mais necessário um diretório `core` para armazenar os arquivos `MY_` utilizados para substituir componentes nativos;

- Não é mais necessário ter classes `MY_` dentro do diretório `Libraries` para estender ou substituir componentes;
- Crie as classes que desejar e adicione os serviços apropriados em `app/Config/Services.php` para carregar seus componentes em vez dos componentes padrões.

Ao se manter atento a essas informações, o processo de atualização da versão 3.x para a 4.x será tranquilo. Lembre-se de que aqui estou falando sobre questões referentes à estrutura padrão do CodeIgniter. Caso sua aplicação utilize uma estrutura diferente do padrão adotado pelo framework, pode haver mudanças nesse processo que vão variar conforme a estrutura utilizada.

SUGESTÃO

Só faça a migração de aplicações da versão 3.x para a 4.x quando realmente for necessário, e antes da migração certifique-se de que está familiarizado com a nova versão do CodeIgniter e que entendeu bem como ele funciona após a reformulação.

Capítulo 4
Possíveis problemas na instalação

Durante o processo de instalação, pode acontecer de você encontrar alguns problemas e, para ajudar a solucioná-los, segue uma lista dos problemas mais comuns que já pude ver e resolver.

Para testar sua aplicação após a instalação você pode utilizar o comando `php spark serve` no terminal. Ele permitirá o acesso à aplicação através do endereço `http://localhost:8080` e você deverá ver uma imagem como a exibida a seguir caso a instalação tenha ocorrido sem problemas.

Mas lembre-se, o comando `php spark serve` não processa o arquivo `.htaccess` e isso pode fazer com que você tenha problemas ao acessar URLs na sua aplicação.

Para testes mais profundos e principalmente para o desenvolvimento de aplicações, use um servidor web como Apache, NGINX ou IIS.

CodeIgniter

Home Docs Community Contribute

Welcome to CodeIgniter 4.0.2

The small framework with powerful features

About this page

The page you are looking at is being generated dynamically by CodeIgniter.

If you would like to edit this page you will find it located at:

```
app/Views/welcome_message.php
```

The corresponding controller for this page can be found at:

```
app/Controllers/Home.php
```

Go further

Learn

The User Guide contains an introduction, tutorial, a number of "how to" guides, and then reference documentation for the components that make up the framework. Check the User Guide !

Discuss

CodeIgniter is a community-developed open source project, with several venues for the community members to gather and exchange ideas. View all the threads on CodeIgniter's forum, or chat on Slack !

Contribute

CodeIgniter is a community driven project and accepts contributions of code and documentation from the community. Why not join us ?

Page rendered in 0.1462 seconds

Environment: production

© 2020 CodeIgniter Foundation. CodeIgniter is open source project released under the MIT open source licence.

Figura 4.1: Página principal da instalação realizada com sucesso

URLs não funcionam

Caso você esteja tentando acessar uma URL (http://localhost:8080/usuario/jonathan) e ela não funciona, faça um teste adicionando `index.php` logo após o host (`http://localhost:8080/usuario/jonathan`), se dessa forma a página for exibida, então pode ser que seu arquivo `.htaccess` esteja configurado de forma incorreta, ou a extensão `mod_rewrite` do Apache está desativada ou nem chegou a ser instalada no servidor.

No caso de servidores compartilhados, a extensão `mod_rewrite` na maioria deles já vem instalada e ativa por padrão, mas se você está utilizando algum tipo de serviço onde você tem autonomia para configurar o servidor, então é necessário verificar essas configurações.

Só carrega a página padrão do CodeIgniter

Se mesmo com suas próprias views criadas e sendo chamadas corretamente o CodeIgniter mostrar apenas a página padrão, pode ser que o servidor não suporte a variável `REQUEST_URI` necessária.

A primeira coisa a se fazer para verificar esse problema é abrir o arquivo `app/Config/App.php` , localizar as informações do protocolo URI e ver se está configurado corretamente:

```
public $indexPage = 'index.php';
```

Se ainda assim não funcionar, então você deve forçar a inserção de um ponto de interrogação (?):

```
public $indexPage = 'index.php?';
```

Uma página 'Ops!' está sendo exibida

Essa página é exibida quando um erro de processamento que impede o carregamento correto da página acontece, e ela é exibida apenas em ambiente de produção, uma vez que nesse ambiente o *debug* da aplicação geralmente está desativado.

Você pode alterar o ambiente no arquivo `.env` para mudar da página `Ops!` para uma página com a exibição de erros e assim poder identificar o problema e solucioná-lo, mas é preciso lembrar de voltar a configuração para ambiente de produção após corrigir o problema.

Uma outra forma de localizar o erro sem precisar mudar a configuração do ambiente é através dos arquivos de log que ficam armazenados em `writable/logs`.

Capítulo 5
Boas práticas de segurança

Segurança de aplicações web é algo que deve ser levado a sério, e o CodeIgniter incorpora vários recursos e técnicas para ajudar você a aplicar práticas de segurança nas suas aplicações.

A equipe de desenvolvimento do framework busca seguir ao máximo as recomendações do Open Web Application Security Project (OWASP) para que suas aplicações se tornem cada vez mais confiáveis e seguras.

A seguir você verá uma lista com dicas da OWASP, identificando as principais vulnerabilidades para aplicações web, além de uma breve descrição, as recomendações da OWASP e as disposições do CodeIgniter para solucionar o problema.

Site Oficial da OWASP: https://www.owasp.org/

5.1 Injection

Uma injection nada mais é do que uma inserção inadequada de dados, sejam eles parciais ou completos, por meio de dados de entrada do usuário para o aplicativo.

Os vetores de ataque incluem SQL, XML, ORM, *overflow* de código e de buffer.

Recomendações da OWASP

- **Apresentação:** definir corretamente o tipo de conteúdo, caracteres e localização;

- **Submissão:** validar os campos e fornecer feedback para o usuário;
- **Controlador:** fazer a limpeza dos dados de entrada, validar as entradas utilizando os conjuntos de caracteres corretos;
- **Model:** consultas parametrizadas.

Disposições do CodeIgniter

- Biblioteca HTTP para filtragem de campos de entrada e metadados;
- Biblioteca de validação de formulários.

5.2 Autenticação fraca e gerenciamento de sessões

Autenticação ou gerenciamento de sessões inadequados podem levar o usuário a obter privilégios que não deveria.

Recomendações da OWASP

- **Apresentação:** validar autenticação e regras, enviar token CSRF com formulários;
- **Design:** utilizar somente o gerenciamento de sessões embutido;
- **Controlador:** validar usuário, regras, token CSRF;
- **Model:** validar regras;
- **Dica:** considerar o uso de um administrador de solicitações.

Disposições do CodeIgniter

- Biblioteca de sessões;
- A biblioteca HTTP fornece validação de CSRF;
- Facilidade para adicionar autenticação de terceiros.

5.3 XSS (Cross Site Scripting)

Validação de entrada insuficiente, onde o usuário pode adicionar conteúdo malicioso a um site e isso afetar os usuários quando o acessarem.

Recomendações da OWASP

- **Apresentação:** codificar todos os dados do usuário ao serem exibidos, definir restrições de entrada;
- **Controlador:** validar entradas;
- **Dica:** processar apenas dados confiáveis, não armazenar conteúdo HTML codificado no banco de dados.

Disposições do CodeIgniter

- Função `esc` ;
- Biblioteca de validação de formulário.

5.4 Referências diretas a objetos inseguros

Referências a objetos inseguros ocorrem quando uma aplicação fornece acesso direto a objetos com base nas informações de entrada fornecidas pelo usuário. Como resultado dessa vulnerabilidade, os invasores podem ignorar a autorização e acessar recursos diretamente no sistema, como registros ou arquivos de banco de dados.

Recomendações da OWASP

- **Apresentação:** não expor dados internos, usar mapas de referências aleatórios;
- **Controlador:** obter dados de fontes confiáveis ou mapas de referências aleatórios;

- **Model:** validar regras do usuário antes de atualizar os dados.

Disposições do CodeIgniter

- Biblioteca de validação de formulário;
- Facilidade para adicionar autenticação de terceiros.

5.5 Configuração de segurança inadequada

A configuração de segurança inadequada na arquitetura de uma aplicação pode levar a erros capazes de comprometer a segurança de toda a arquitetura.

Recomendações da OWASP

- **Apresentação:** fortalecer a segurança do servidor web onde a aplicação se encontra, usar HTTPS;
- **Controlador:** fortalecer a segurança do servidor web onde a aplicação se encontra, proteger arquivos e estruturas XML;
- **Model:** proteger o servidor de banco de dados.

Disposições do CodeIgniter

- Verificações de integridade durante a inicialização.

5.6 Exposição de dados confidenciais

Dados confidenciais devem ser protegidos quando transmitidos pela rede. Esses dados podem incluir credenciais de usuários e cartões de créditos. Como regra geral, se os dados devem ser protegidos quando armazenados, eles também devem ser protegidos durante a transmissão.

Recomendações da OWASP

- **Apresentação:** use TLS 1.2, use codificações e hashes fortes, não envie chaves ou hashes para o navegador;
- **Controlador:** use codificações e hashes fortes;
- **Model:** exigir comunicação criptografada forte com o servidor.

Disposições do CodeIgniter

- Chaves de sessão armazenadas criptografadas.

5.7 CSRF (Cross Site Request Forgery)

O CSRF é um ataque que força um usuário final a executar ações indesejadas em uma aplicação no qual ele está atualmente autenticado.

Recomendações da OWASP

- **Apresentação:** validar usuários e regras, enviar tokens CSRF;
- **Controlador:** validar usuários e regras, enviar tokens CSRF;
- **Model:** validar regras.

Disposições do CodeIgniter

- A biblioteca HTTP fornece validação CSRF.

5.8 Usando componentes com vulnerabilidades conhecidas

Muitas aplicações possuem vulnerabilidades conhecidas e estratégias de ataque que podem ser exploradas para obter controle

remoto ou explorar dados.

Recomendações da OWASP

Não utilize nenhuma dessas aplicações.

Disposições do CodeIgniter

- Bibliotecas de terceiros incorporadas à aplicação devem ser examinadas.

5.9 Redirecionamentos e encaminhamentos não validados

Os problemas de redirecionamento costumam ocorrer devido a lógica comercial defeituosa ou código acionável injetado, capaz de redirecionar o usuário de maneira inadequada dentro da aplicação.

Recomendações da OWASP

- **Apresentação e Controlador:** não utilizar redirecionamento de URL, usar referências indiretas aleatórias;
- **Model:** validar regras.

Disposições do CodeIgniter

- A biblioteca HTTP oferece recursos para lidar com redirecionamentos e encaminhamentos;
- A biblioteca Session fornece *flashdata*, que são dados transmitidos via sessão e que estão disponíveis apenas para a próxima solicitação, sendo apagados automaticamente.

Arquitetura

Agora que você já tem as noções básicas sobre o CodeIgniter 4, as mudanças em relação à versão 3.x e como fazer a instalação dele, é hora de entender a arquitetura da nova versão.

Nos próximos capítulos você verá questões como a estrutura da aplicação, o modelo MVC, Autoloading, Services, Requisições HTTP e entre outras coisas que vão lhe ajudar no desenvolvimento de aplicações com o CodeIgniter 4.

Capítulo 6
Estrutura da aplicação

Para que você possa extrair o melhor da performance e da produtividade que o CodeIgniter pode lhe proporcionar, é preciso entender bem como ele está estruturado e o que você pode alterar nessa estrutura padrão para atender à necessidade da sua aplicação.

Toda instalação da versão 4 possui cinco diretórios bases: `/app`, `/system`, `/public`, `/writable`, `/tests`. Cada um desses diretórios desempenha uma função específica dentro da aplicação.

Se a instalação foi feita de forma manual, através do download dos arquivos, um sexto diretório, `/user_guide`, aparecerá e nele estará a documentação da versão do framework que acabou de instalar. Também pode acontecer do diretório `/system` não estar sendo exibido, caso você tenha feito uma instalação via *Composer* a partir de `codeigntier4/appstarter`.

Veja a seguir os detalhes sobre cada um desses diretórios.

/app

É o diretório onde estará todo o código da sua aplicação. Ele possui uma estrutura com subdiretórios para que o código da aplicação

fique organizado.

- `/app`
  - `/Config` : armazena os arquivos de configuração da aplicação;
  - `/Controllers` : armazena os controladores da aplicação;
  - `/Database` : armazena as `migrations` de bancos de dados e os arquivos `seed` ;
  - `/Filters` : armazena os filtros que podem ser executados antes e depois do controlador;
  - `/Helpers` : armazena `collections` e funções independentes da aplicação;
  - `/Language` : armazena os arquivos de tradução da aplicação, dando suporte a diversos idiomas;
  - `/Libraries` : armazena classes úteis à aplicação;
  - `/Models` : armazena os `models` que trabalharão junto com o banco de dados representando as entidades da aplicação;
  - `/ThirdParty` : armazena bibliotecas de terceiros que podem ser utilizadas na aplicação;
  - `/Views` : armazena os arquivos com a estrutura HTML das páginas que compõem a aplicação.

Todos os arquivos desse diretório estão no namespace `App` , mas você pode alterá-lo em `app/Config/Constants.php` .

/system

É nesse diretório que ficam armazenados os arquivos que compõem a estrutura do framework. Você **nunca deve modificar** os arquivos desse diretório.

Caso precise alterar algo em relação ao funcionamento padrão do framework, você deve criar novas classes e estender a partir da classe padrão, de modo a prover a funcionalidade necessária à aplicação. Na parte final do livro você encontrará capítulos onde falo com mais detalhes sobre como estender o framework. Todos os arquivos nesse diretório estão no namespace `CodeIgniter` .

/public

Esse diretório contém os arquivos acessíveis ao navegador utilizado para executar a aplicação. Dessa forma o acesso ao código-fonte da aplicação fica bloqueado para acesso direto.

Ao iniciar uma nova aplicação, seja por instalação manual ou via *Composer*, esse diretório já vem com alguns arquivos: `.htaccess`, `index.php`, `favicon.ico`, `robots.txt`. Os arquivos CSS, JavaScript, imagens e outros necessários à aplicação que possuírem acesso público devem ser armazenados nesse diretório.

> **ATENÇÃO**
>
> Esse diretório deve ser *o diretório raiz* da sua aplicação e o servidor web onde ela estiver hospedada deve estar configurado para apontar para ele.

/writable

É nesse diretório que ficam armazenados os arquivos de cache, log, uploads feitos pelo usuário e outros arquivos que você precise armazenar em tempo de execução. Se você precisar organizar os uploads de arquivos da aplicação, pode criar subdiretórios dentro de `/writable`.

Com esse diretório específico para gravação de arquivos, você aumenta a segurança da sua aplicação, uma vez que pode manter todos os demais diretórios apenas com permissão de leitura.

/tests

Esse diretório está configurado para armazenar os arquivos de testes. No diretório `/tests/_support` você encontra diversas classes

simuladas e utilitários que podem ser utilizados durante a escrita dos testes.

> **ATENÇÃO**
>
> Ao colocar a aplicação no ambiente de produção, o diretório `/_support` não precisa ser enviado para o servidor.

/user_guide

Nesse diretório está armazenada a documentação oficial do CodeIgniter (*User Guide*) e, ao colocar a aplicação em produção, esse diretório não precisa ser enviado.

> **MODIFICANDO O ENDEREÇO DOS DIRETÓRIOS**
>
> Caso você precise alterar a localização dos diretórios da aplicação, certifique-se de informar a localização deles no arquivo `app/Config/Paths.php`.

CAPÍTULO 7
Gerenciamento da aplicação

Fazer o gerenciamento da aplicação no CodeIgniter é bem simples, e isso se dá por sua estrutura de arquivos enxuta e bem organizada.

7.1 Renomeando ou alterando o diretório /app

Se por alguma particularidade da sua aplicação você precisar modificar o nome do diretório `/app` ou mesmo mudá-lo de lugar no servidor, que não seja a raiz do projeto, você deverá abrir o arquivo `/app/Config/Paths.php` e definir o caminho completo do servidor na variável `$appDirectory` ou alterar o nome de `app` para o novo nome dado ao diretório.

```
public $appDirectory = 'caminho/do/diretorio/app/no/servidor';
```

Além disso, você também precisará modificar outros 2 arquivos na raiz do projeto para que estes identifiquem a nova localização do diretório `/app` :

Modificando o arquivo `spark`

Abra o arquivo `spark` que se encontra na raiz do projeto e altere a linha `require '/app/Config/Paths.php';` informando a nova localização do arquivo.

```
require 'caminho/do/diretorio/app/Config/Paths.php';
```

Modificando o arquivo `/public/index. php`

Você também precisará alterar o valor da variável `$pathsPath` no arquivo `/public/index.php`.

```
$pathsPath = FCPATH . 'caminho/do/diretorio/app/Config/Paths.php';
```

7.2 Executando múltiplas aplicações com uma mesma instalação do CodeIgniter

Caso você tenha várias aplicações rodando no servidor e queira utilizar uma única instalação da estrutura do CodeIgniter, basta criar os diretórios das aplicações na raiz do projeto e mover para dentro deles a estrutura padrão da instalação, exceto o diretório `/system`.

Vamos supor que você tenha duas aplicações rodando no servidor: *gestao* e *mkt*. A estrutura ficaria como a seguinte:

- `/gestao`
  - `/app`
  - `/public`
  - `/tests`
  - `/writable`
- `/mkt`
  - `/app`
  - `/public`
  - `/tests`
  - `/writable`
- `/system`

O arquivo `index.php` de cada aplicação aponta para a própria configuração `../app/Config/Paths.php` e a variável `$systemDirectory` dentro do arquivo `/app/Config/Paths.php` de cada aplicação deverá apontar para o diretório compartilhado `/system`.

Algo semelhante ao exemplo a seguir:

```
public $systemDirectory = __DIR__ . '/../../../system';
```

Caso uma das aplicações, ou todas elas, tenha componentes de linha de comando, você também deverá modificar o arquivo `spark` presente dentro do diretório da aplicação, como visto no início do capítulo.

Capítulo 8
Múltiplos ambientes

Geralmente, o processo de desenvolvimento de uma aplicação envolve de dois a três ambientes diferentes: *desenvolvimento*, *testes* e *produção*.

Cada um desses ambientes possui suas particularidades e uma ocorrência muito comum é com relação às mensagens de erro. Nos ambientes de *desenvolvimento* e *testes*, são exibidas mensagens de erro detalhadas, enquanto no ambiente de *produção* essas mensagens são filtradas e com legibilidade no nível de usuário e não no de desenvolvedor.

Também pode acontecer de ser necessário carregar ferramentas e bibliotecas adicionais no ambiente de desenvolvimento que não são necessárias no ambiente de produção. Para ajudar a organizar esses ambientes, o CodeIgniter conta com a constante `ENVIRONMENT`.

8.1 A constante ENVIRONMENT

Essa é a constante responsável por determinar para a aplicação qual o ambiente em que ela está sendo executada. O valor dessa constante é definido na variável `$_SERVER['CI_ENVIRONMENT']`, e caso não seja informado, o valor padrão será `production`.

Existem diversas formas de definir o valor dessa constante, conforme a configuração do servidor utilizado para executar a aplicação.

`.env`

A forma mais simples de definir a variável é através do arquivo `.env` existente na raiz da aplicação.

```
CI_ENVIRONMENT = `development`
```

> Caso você faça a alteração no arquivo e ainda sim ela não seja refletida na aplicação, verifique se o arquivo está nomeado como `env` ou `.env`. Se estiver como `env` mude para `.env`, e caso esteja `.env` verifique se as configurações do servidor Apache estão corretas e o módulo `env_module` ativado.

Apache

Também é possível definir o valor da variável através do arquivo `.htaccess` ou diretamente na configuração do Apache usando `SetEnv`.

```
SetEnv CI_ENVIRONMENT development
```

NGINX

No NGINX você deve passar a variável de ambiente pelos `fastcgi_params` para que ela seja adicionada à variável `$_SERVER`. Isso permitirá que ela funcione em nível de host virtual em vez de utilizar o `.env` para configurá-la para todo o servidor, embora isso funcione bem em um servidor dedicado.

A configuração do NGINX seria algo como:

```
server {
    server_name localhost;
    include     conf/defaults.conf;
    root        /var/www;

    location    ~* \.php$ {
        fastcgi_param CI_ENVIRONMENT "production";
        include conf/fastcgi-php.conf;
    }
}
```

Arquivos de inicialização

Para o CodeIgniter, é necessário que exista um script PHP correspondente ao nome do ambiente e que este se localize em `/app/Config/Boot` .

Esses arquivos podem ser personalizados de acordo com a forma como você desejar que a aplicação funcione no ambiente que está configurando.

O sistema os carrega automaticamente, então sempre que você fizer uma instalação nova do CodeIgniter, os arquivos a seguir serão criados dentro do diretório `/app/Config/Boot` :

- `development.php`
- `production.php`
- `testing.php`

8.2 Efeitos no comportamento padrão da estrutura

Existem alguns locais dentro da estrutura padrão do CodeIgniter em que a constante `ENVIRONMENT` é utilizada e as respostas são de acordo com o ambiente definido na constante:

Exibição de erros

Se você definir a constante para `development` todos os erros do PHP serão renderizados no navegador sempre que eles ocorrerem. Mas se a constante estiver definida para `production` , a exibição de erros será desativada por completo.

Manter a exibição de erros desativada no ambiente de produção é uma boa prática de segurança.

Arquivos de configuração

De forma opcional, você também pode fazer com que os arquivos de configuração da aplicação sejam carregados de acordo com o ambiente configurado. Essa é uma prática útil para, por exemplo, gerenciar APIs em ambientes diferentes, conexões diferentes a bancos de dados, entre outras.

No próximo capítulo você verá com mais detalhes como trabalhar com os arquivos de configuração.

Capítulo 9
Arquivos de configuração

Os arquivos de configuração são os responsáveis por definir muitas das formas de funcionamento da aplicação. No CodeIgniter, eles mantêm uma classe contendo as configurações com propriedades públicas.

Não existe uma única classe a ser utilizada para acessar suas configurações. Você simplesmente cria uma instância da classe e todas as suas configurações ficam disponíveis.

O diretório em que os arquivos de configuração ficam armazenados é `/app/Config`.

9.1 Acessando os arquivos de configuração

Para acessar os arquivos de configuração em suas classes você pode criar uma nova instância ou utilizar a função `config`. Como todas as propriedades são públicas, você as acessa como qualquer outra propriedade:

```
//CRIAÇÃO MANUAL
$config = new \Config\Email();

//CRIAÇÃO COM A FUNÇÃO config
$config = config('Email', false);

//CRIAÇÃO COM UMA INSTÂNCIA COMPARTILHADA
$config = config('Email');

//FUNÇÃO config COM namespace
$config = config('Config\\Email');
```

```
//ACESSANDO AS CONFIGURAÇÕES COMO PROPRIEDADES DE CLASSE
$SMTPHost = $config->SMTPHost;
```

Caso nenhum `namespace` seja fornecido, o arquivo será procurado em todos os `namespaces` disponíveis dentro do diretório `/app/Config`.

Utilizar o `namespace Config` na sua aplicação fornecerá um melhor desempenho, pois evitará que seja feita uma busca no diretório de arquivos para identificar qual é o `namespace` a ser utilizado.

9.2 Criando arquivos de configuração

Se a sua aplicação tiver a necessidade de arquivos de configuração específicos, você pode criá-los dentro do diretório `/app/Config` e, em seguida, escrever o código da classe com as propriedades públicas que representam suas configurações.

A classe deve estender `\CodeIgniter\Config\BaseConfig` para garantir que possa receber configurações específicas do ambiente.

```
<?php namespace Config

use CodeIgniter\Config\BaseConfig;

class Livro extends BaseConfig
{
    public $autor = 'Jonathan Lamim';
    public $editora = 'Casa do Código';
}
```

9.3 Trabalhando com ambientes diferentes

Como foi citado no capítulo anterior, uma aplicação pode rodar em ambientes diferentes (*desenvolvimento*, *testes* e *produção*) e,

considerando que cada ambiente possui suas particularidades no que diz respeito às configurações, você deve criar arquivos `.env` para cada ambiente de modo a controlar de forma mais simples as configurações da aplicação.

Quando sua aplicação for executada, o arquivo `.env` será carregado de forma automática e as variáveis contidas neles, inseridas no ambiente.

As variáveis contidas no arquivo `.env` podem ser recuperadas através de `getenv()`, `$_SERVER` e `$_ENV`. Das três, a função `getenv()` é a mais recomendada, uma vez que ela não diferencia maiúsculas de minúsculas.

9.4 Variáveis de aninhamento

Para agilizar o processo de desenvolvimento e escrita do arquivo `.env` você pode reutilizar variáveis já especificadas envolvendo o nome da variável com `{}`.

```
BASE_DIR = "/diretorio/raiz/do/projeto"
CACHE_DIR = "${BASE_DIR}/cache"
TMP_DIR = "${BASE_DIR}/tmp"
```

9.5 Variáveis com namespace

Em alguns momentos, você terá variáveis com o mesmo nome. Quando isso acontecer, o sistema não saberá distinguir o valor correto a ser retornado. Para se proteger contra esse problema, utilize um `namespace` para as variáveis.

Variáveis com `namespace` são definidas da mesma maneira, adicionando apenas o `namespace` como prefixo e um ponto (.) para

separar `namespace` e o nome da variável.

```
// variáveis sem namespace
autor = 'Jonathan Lamim'
editora = 'Casa do Código'

//variáveis com namespace
livro.autor = 'Jonathan Lamim'
livro.editora = 'Casa do Código'
```

9.6 Incorporando variáveis de ambiente em uma configuração

Ao instanciar um arquivo de configuração, qualquer variável de ambiente com `namespace` é mesclada nas propriedades do objeto de configuração.

Se o prefixo de uma variável com `namespace` corresponder exatamente ao nome de uma classe de configuração, com distinção entre maiúsculas e minúsculas, a parte à direita do ponto (.) será tratada como um nome de propriedade de configuração.

Caso o nome da variável corresponda a uma propriedade de configuração existente, o valor da variável de ambiente substituirá o valor correspondente no arquivo de configuração. Se não houve correspondência, as propriedades de configuração permanecerão inalteradas.

A mesma regra vale para um *short prefix*, que é o nome dado ao caso quando o prefixo da variável de ambiente corresponde ao nome da classe de configuração convertida em minúsculas.

```
//variáveis no arquivo .env
Livro.autor = 'Jonathan Lamim'
Livro.editora = 'Casa do Código'
```

```php
//variáveis no arquivo de configuração
<?php namespace Config

use CodeIgniter\Config\BaseConfig;

class Livro extends BaseConfig
{
    public $autor = 'Jonathan Lamim';
    public $editora = 'Casa do Código';
}
```

9.7 Tratando variáveis de ambiente como arrays

Uma variável de ambiente pode ser tratada como um array quando o prefixo corresponder à classe de configuração e o nome da variável for separado por ponto (.) em relação ao próximo identificador.

```
// variável com namespace
Livro.autor = 'Jonathan Lamim'

// variáveis com namespace em formato de matriz
Livro.autor.nome = 'Jonathan Lamim'
Livro.autor.cidade = 'Vila Velha - ES'
```

Fazendo referência ao objeto de configuração `Livro` , o mesmo ficaria da seguinte forma:

```php
$autor['nome'] = 'Jonathan Lamim';
$autor['cidade'] = 'Vila Velha - ES';
```

Qualquer outro elemento da propriedade `$autor` não seria alterado.

9.8 Registrars

Também é possível que um arquivo de configuração especifique registradores, chamados no CodeIgniter de *registrars*, que são classes capazes de fornecer propriedades de configurações adicionais para a aplicação.

Para atuarem como um registrador, as classes identificadas devem ter uma função estática cujo nome é igual à classe de configuração e devem retornar um array associativo de configurações de propriedades.

Ao instanciar o objeto de configuração, ele faz um loop sobre as classes definidas em `$registrars` e para cada uma dessas classes que contêm o nome de método correspondente à classe de configuração, ele executará o método e incorporará todas as propriedades retornadas da mesma forma como é feito com as variáveis definidas com `namespace`.

Veja a seguir um exemplo da classe de configuração:

```
<?php namespace App\Config;

use CodeIgniter\Config\BaseConfig;

class LivroConfig extends BaseConfig
{
    public $autor         = "Jonathan Lamim";
    public $editora       = "Casa do Código";
    protected $registrars = [
        '\App\Models\Livro';
    ];
}
```

Agora veja um exemplo do model informado em `$registrars`:

```
<?php namespace App\Models;

class Livro
{
    public static function LivroConfig()
    {
```

```
        return ['editora' => "CDC", 'edicao' => 1];
    }
}
```

Ao instanciar a classe `LivroConfig` ela terá as duas propriedades declaradas, porém o valor da propriedade `editora` será substituído, uma vez que o model `Livro` está sendo tratado como um registrador.

Esse é um recurso bastante interessante de se utilizar, pois ele dá a possiblidade de alterar as configurações padrões feitas no arquivo de configuração por informações dinâmicas, a partir de informações vindas do banco de dados ou outras fontes diretamente no model, em tempo de execução.

Capítulo 10
Modularização

O CodeIgniter tem suporte à modularização de código para proporcionar ao desenvolvedor a criação de código reutilizável. Os módulos são tipicamente centrados em um assunto específico e podem ser vistos como *miniaplicações* dentro da aplicação principal.

Controladores, *models*, *views*, arquivos de configuração, arquivos de idiomas, qualquer um dos tipos de arquivos padrão dentro da estrutura do framework tem suporte à modularização. Os módulos da aplicação a ser desenvolvida podem conter quantos desses arquivos forem necessários para seu funcionamento.

10.1 Namespaces

Nessa nova versão do CodeIgniter, os *namespaces* tornaram-se o elemento principal da funcionalidade dos módulos, vindo do carregamento automático compatível com a PSR4 (Autoloader).

Mesmo sabendo que qualquer código possa fazer uso da PSR4 e dos namespaces, a principal forma de obter o máximo de proveito dos módulos é atribuir um namespace ao seu código e adicioná-lo ao arquivo `/app/Config/Autoload.php` .

Vamos supor que você esteja desenvolvendo um site institucional com gerenciador de conteúdo personalizado para a empresa ABC e o cliente queira um blog junto ao site. Você pode criar um diretório na raiz da aplicação com o nome da empresa (ABC) para armazenar todos os módulos da aplicação, ficando com uma estrutura similar à apresentada a seguir:

- `/abc`

- /app
- /public
- /system
- /tests
- /writable

Abra o arquivo `/app/Config/Autoload.php` e adicione o namespace `Abc` à propriedade da matriz PSR4:

```
public $psr4 = [
    'Abc' => ROOTPATH.'abc'
];
```

Feita a configuração, qualquer arquivo localizado dentro do diretório `abc` pode ser acessado utilizando o namespace `Abc` .

Essa simples configuração é responsável por 80% do que é necessário para o funcionamento dos módulos, ou seja, é importante você estar familiarizado com o uso de namespaces para extrair o máximo do potencial que o CodeIgniter 4 tem a oferecer.

Uma outra informação importante que você precisa ter sempre em mente é que dentro de cada módulo haverá uma estrutura de diretórios e arquivos similar à estrutura do diretório principal da aplicação:

- /abc
  - /Blog
    - /Config
    - /Controllers
    - /Database
      - /Migrations
      - /Seeds
    - /Helpers
    - /Language
      - /en
    - /Libraries
    - /Models

- `/Views`

Não existe nenhuma obrigatoriedade para que você utilize essa estrutura, você tem liberdade para estruturá-la da forma que melhor se adequar ao módulo, eliminando os diretórios desnecessários.

> **SUGESTÃO**
>
> Procure manter a estrutura de diretórios e arquivos dos módulos em similaridade com a estrutura padrão do framework, assim fica mais fácil para localizar arquivos e, principalmente, para aplicar futuras atualizações.

10.2 Auto-discovery

Por diversas vezes em uma aplicação você precisará especificar o namespace completo dos arquivos que deseja incluir, mas no CodeIgniter você pode aplicar uma configuração para simplificar a integração dos módulos em suas aplicações, identificando de forma automática muitos tipos diferentes de arquivos, incluindo *Eventos*, *Registradores*, *Arquivos de rotas* e *Serviços*.

Essa configuração é feita no arquivo `/app/Config/Modules.php`. O sistema de identificação automática funciona fazendo uma varredura de todos os namespaces da PSR4 que foram definidos em `/app/Config/Autoload.php`, buscando por diretórios e arquivos familiares.

Seguindo o exemplo apresentado anteriormente para o caso da empresa *ABC*, precisamos fazer um pequeno ajuste para que os arquivos possam ser encontrados.

Cada módulo dentro do namespace `Abc` precisa ter seu próprio namespace definido. Por exemplo, `Abc` seria alterado para `AbcBlog`,

assim, uma vez definida a pasta do módulo, o processo de identificação buscará um arquivo de rotas, por exemplo `/abc/Blog/Config/Routes.php`, como se fosse outra aplicação.

Ativando/Desativando a identificação automática

É possível você ativar ou desativar toda a descoberta automática no sistema com a variável de classe `$enabled`. Se for definida como `false`, ela desativa todas as descobertas, otimizando o desempenho, mas negando os recursos especiais de seus módulos.

Especificando itens da identificação

Através da opção `$activeExplorers` você pode especificar quais itens são identificados automaticamente. Se o item não estiver presente, não será feita nenhuma identificação automática para ele, mas os demais contidos no array ainda serão identificados.

Identificação automática e Composer

Os pacotes que foram instalados via *Composer* são descobertos por padrão, requerendo apenas que o namespace conhecido pelo *Composer* seja um namespace PSR4. Namespaces PSR0 (Autoloading Standard) não serão detectados.

Se você não quer que todos os diretórios conhecidos do *Composer* sejam verificados ao localizar arquivos, basta desativá-los editando a variável `$discoverInComposer` localizada no arquivo `/app/Config/Modules.php`.

DIFERENÇA ENTRE NAMESPACES PSR0 E PSR4 Namespaces PSR0 possuíam recursos compatíveis com nomes de classe no estilo PEAR. Na PSR4 isso foi eliminado, suportando apenas código que utilize namespace. Outro detalhe importante é que a PSR4 não o obriga a ter todo o namespace como uma estrutura de diretórios, apenas a parte que segue o ponto de ancoragem. A PSR0 foi marcada como *deprecated* em 21/10/2014 e foi recomendada a utilização da PSR4 em substituição a ela.

10.3 Trabalhando com arquivos

Até agora você viu como realizar as configurações para utilizar a modularização dentro da sua aplicação. Nesta parte do capítulo você verá como arquivos de controladores, models, views, idiomas podem ser utilizados dentro dos módulos.

Como teremos capítulos específicos ao longo do livro para falarmos sobre controladores, models, views e outros arquivos, vou ser mais objetivo nessa parte para que você possa entender a aplicação de cada arquivo dentro do módulo, facilitando sua compreensão quanto à modularização de aplicações com CodeIgniter.

Rotas

As rotas são os caminhos utilizados pelo CodeIgniter para processar as requisições da sua aplicação, e são verificadas de forma automática nos módulos. A instância `$routes` já está definida para a aplicação por padrão, ou seja, se tentar redefinir essa classe, a aplicação apresentará erros.

As rotas trabalham com um tipo de ligação direta com os controladores, uma vez que os métodos inseridos neles podem responder para rotas da aplicação.

Controladores

Controladores são os responsáveis por executarem o conjunto de operações que darão ao usuário o resultado que ele está buscando com a aplicação. Eles respondem às rotas e ficam localizados no diretório `/app/Controllers` ou no caso de modularização, na estrutura de diretório `/abc/app/Controllers` .

Quando os controladores estão fora do diretório principal da aplicação `/app/Controllers` eles não podem ser roteados de forma automática através da detecção de URI, precisando ser identificados no próprio arquivo de configuração de rotas, `/app/Confis/Routes.php` .

```
$routes->get('blog', 'Abc\Blog\Controllers\Blog::index');
```

Considerando que cada módulo pode ter diversas rotas, defini-las com essa notação geraria um volume considerável de código. Mas para facilitar isso, você pode utilizar o **roteamento em grupo** - recurso muito útil para rotear módulos.

```
$routes->group('blog', ['namespace' => 'Abc\Blog\Controllers'],
function($routes)
{
    $routes->get('/', 'Blog::index');
});
```

Esse exemplo faz a mesma coisa que o apresentado anteriormente, porém você tem uma estrutura mais organizada e um código mais legível.

Arquivos de configuração

Não há nenhuma mudança especial a ser feita nos arquivos de configuração para trabalhar com módulos na aplicação. Eles continuam sendo *namespaced* e carregados como comandos:

```
$config = new \Abc\Blog\Config\Blog();
```

Eles são identificados de forma automática sempre que a função `config` estiver disponível.

Migrations

Os arquivos de migração também são identificados de forma automática a partir dos namespaces já definidos. Todas as *migrations* encontradas em todos os namespaces sempre serão executadas.

Seeds

Os arquivos de propagação podem ser utilizados na linha de comando e chamados a partir de outros arquivos de propagação, desde que seja fornecido o namespace completo. Ao chamar um arquivo de propagação via linha de comando, será necessário utilizar barras invertidas duplas:

```
php public/index.php migrations seed
Abc\\Blog\\Database\\Seeds\\TestPostSeeder
```

Helpers

Os helpers são bibliotecas auxiliares que também são identificadas automaticamente a partir de namespaces definidos quando o método `helper()` é utilizado. Lembre-se de que para serem identificados de forma automática precisam estar dentro do diretório `/app/Helpers`.

```
helper('blog');
```

Arquivos de idioma

Os arquivos de idioma são identificados de forma automática através dos namespaces definidos ao utilizar o método `lang()`. Assim como nos helpers, eles precisam seguir a mesma estrutura de diretórios da aplicação principal.

Mais adiante no decorrer do livro veremos mais detalhes sobre o uso de arquivos de idioma.

Libraries

As libraries são bibliotecas sempre instanciadas pelo nome da classe e compõem a aplicação com recursos pontuais.

```
$lib = new \Abc\Blog\Libraries\BlogLib();
```

Mais adiante no decorrer do livro veremos mais detalhes sobre o uso das bibliotecas.

Models

O model na versão 4 do CodeIgniter possui mais funcionalidades, incluindo conexão automática ao banco de dados, CRUD básico, validação no próprio model e paginação automática de registros. Eles são sempre instanciados pelo nome da classe.

```
$model = new \Abc\Blog\Models\PostModel();
```

Views

As views são a parte na qual o usuário recebe as respostas da aplicação de forma visual e são carregadas utilizando o namespace da classe:

```
echo view('Abc\Blog\Views\index');
```

Capítulo 11
MVC

Organizar o código da sua aplicação é fundamental para que você possa ter maior desempenho tanto no processo de desenvolvimento quanto da própria aplicação no ambiente de produção.

Assim como a maioria dos frameworks PHP, o CodeIgniter utiliza o padrão MVC para organizar os arquivos, mantendo dados, apresentação e fluxo da aplicação com partes separadas.

Neste capítulo faremos uma introdução sobre cada parte do padrão MVC e no decorrer do livro será apresentada cada parte com mais detalhe dentro da abordagem do CodeIgniter 4.

Compreendendo o MVC

Existem diversas visões sobre os papéis de cada parte desse padrão, portanto nas linhas a seguir você verá os papéis dentro do framework, para que a compreensão de seu funcionamento seja a maior possível.

Os models (M) são responsáveis por gerenciar os dados da aplicação, ajudando a organizar regras específicas, além do que já se faz normalmente com dados (ler, gravar, listar, atualizar).

As views (V) são arquivos mais simples, onde os dados são organizados para a renderização no browser do usuário. O ideal é que as views contenham apenas as informações a serem exibidas, sem nenhuma lógica complexa, exceto aquelas necessárias para a exibição das informações.

Os controladores (C) são basicamente o conjunto de regras de negócio da aplicação, operando como um mediador entre a view e o model.

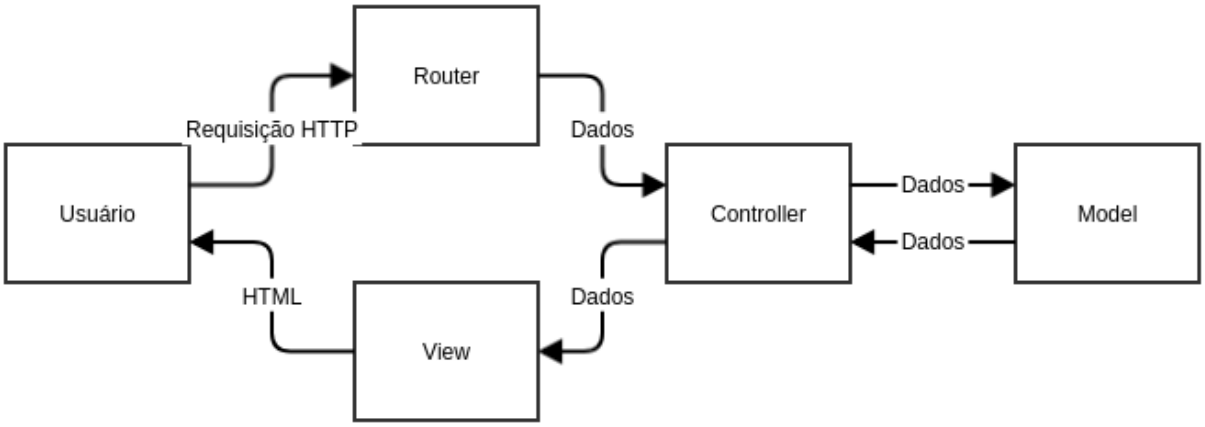


Figura 11.1: Representação do fluxo da comunicação no padrão MVC

11.1 Model (M)

Os models têm como finalidade prover os dados para a aplicação a partir de solicitações do controlador. O trabalho deles é dividido em duas partes: impor as regras de negócio aos dados nos processos de inserção e extração no banco de dados; e lidar com as requisições feitas pelo controlador para salvar, atualizar, excluir ou recuperar dados no banco de dados.

É comum confundir as regras que devem ficar no controlador e as que devem ficar no model, mas uma boa forma de resolver essa confusão é ter em mente que no model devem estar todas as regras relacionadas a restrições ou requisitos de dados.

Por exemplo, se você precisar recuperar as informações do perfil de um usuário específico, o model deverá ser o responsável por implementar a regra que limita a busca por esse perfil, seja por um ID, email ou qualquer outra informação do usuário.

Outro tipo de regra que deve estar no model é a normalização dos dados antes de salvá-los no banco de dados. Se você precisa aplicar formatações nos dados, como datas, remoção de espaços

antes e depois de strings ou outras, é no model que você implementará essas regras.

Fazendo dessa forma você evita código repetido em diversos controladores que fazem uso das mesmas informações, otimizando o processo de desenvolvimento e também a aplicação.

Os models são armazenados no diretório `/app/Models`, mas nada impede que você os organize em outro diretório, desde que faça as devidas configurações, como já foi apresentado neste livro.

11.2 View (V)

As views são os arquivos mais simples dentro da composição de uma aplicação no CodeIgniter, pois são compostos na maior parte por código HTML e pouquíssimo código PHP. O recomendado é que as views contenham o mínimo possível de código PHP, para que a organização e o conceito original do padrão MVC sejam mantidos.

Elas recebem os dados em forma de variáveis devidamente processadas pelo controlador para que sejam exibidas com facilidade, usando um `echo` ou até mesmo pseudo-variáveis que são suportadas pelo CodeIgniter.

Os arquivos contendo as views da aplicação ficam armazenados no diretório `/app/Views`, mas esse local pode ser alterado desde que as devidas configurações sejam realizadas para apontar a nova localização desses arquivos, como já vimos em capítulos anteriores.

No CodeIgniter uma boa prática é que para cada controlador da aplicação seja criado um diretório em `/app/Views` e que os arquivos tenham o nome do respectivo método no controlador. Isso agiliza o processo de localização das views dentro da aplicação.

Por exemplo, se na sua aplicação você precisa fazer o controle de usuários e tem um controlador chamado `Usuario` e um método chamado `perfil` que deve exibir as informações do perfil de um usuário, uma boa prática é ter a seguinte estrutura:

- `/app`
  - `/Views`
    - `/Usuario`
      - `Perfil.php`

Essa organização, quando se torna um hábito, aumenta a produtividade no processo de desenvolvimento da aplicação e o nível de organização do projeto.

11.3 Controller (C)

Os controladores são as bases da regra de negócios da aplicação e desempenham 2 papéis diferentes. O primeiro é receber as informações do usuário e determinar o que fazer com elas. Ao definir o que fazer com elas o controlador pode iniciar uma comunicação com um model para salvar dados ou solicitar dados que o usuário deseja visualizar. Nesse processo ele também pode chamar outras classes da aplicação para poder completar o processamento das informações a serem retornadas ao usuário.

O segundo papel do controlador é lidar com tudo o que diz respeito às solicitações HTTP (redirecionamentos, autenticação, segurança, codificação etc.).

No geral, o controlador é onde você garante que os usuários estão utilizando a aplicação e recebendo as informações que desejam.

Eles são armazenados no diretório `/app/Controllers` mas, assim como os models e as views, podem ser armazenados em outro

diretório, desde que os apontamentos e namespaces estejam devidamente configurados.

Capítulo 12
Autoloading

Toda aplicação consiste em um número considerável de classes e bibliotecas distribuídas em diretórios variados dentro da estrutura da aplicação que o fazem funcionar. Manter o controle de onde cada arquivo está localizado e chamar cada um deles utilizando `require()` é bastante dispendioso e aumenta a chance de erros. É para ajudar nesse processo que entra o *autoloading*, ou carregamento automático de arquivos.

O CodeIgniter possui um autoloading bastante flexível que pode ser utilizado com pouca configuração. Ele é capaz de localizar classes individuais sem namespace, classes com namespace adequadas à PSR4 e até localizar classes em diretórios comuns da aplicação.

O autoloading nativo é capaz de funcionar bem sozinho e acompanhado de outros autoloadings, como o Composer, se necessário. Por serem registrados por meio do `spl_autoload_register`, trabalham em sequência e não se interpõem.

No CodeIgniter o autoloading está sempre ativo, sendo registrado com `apl_autoload_register()` no início da execução da estrutura, e visando melhorar o desempenho, os principais componentes do CodeIgniter foram adicionados ao mapa de classes.

Configuração

A configuração inicial é feita em `/app/Config/Autoload.php`. Esse arquivo contém os dois principais arrays: um para o mapa de classe, outro para os namespaces compatíveis com a PSR4.

Namespaces

A forma recomendada para organizar suas classes é criar um ou mais namespaces para os arquivos da aplicação. Isso é ainda mais

importante quando estiver lidando com classes relacionadas a regras de negócio, entidades etc.

O array PSR4 no arquivo de configuração permite mapear os namespaces para o diretório em que essas classes podem ser encontradas. A chave de cada linha é o próprio namespace.

```
$psr4 = [
    'Config'      => APPPATH . 'Config',
    'App'         => APPPATH,
    'CodeIgniter' => SYSTEMPATH,
];
```

Por padrão o diretório da aplicação é um namespace `App`. Embora você não seja obrigado a utilizar namespaces nos controladores, bibliotecas ou models dentro do diretório da aplicação, utilizá-los melhorará a performance da aplicação otimizando o processo de autoloading.

O namespace da aplicação pode ser alterado em `/app/Config/Constants.php` e ter o novo valor definido na configuração `APP_NAMESPACE`.

```
define ('APP_NAMESPACE', 'App');
```

Mas atenção, você precisará modificar todos os arquivos existentes que fazem referência ao namespace que foi modificado.

> Os arquivos de configuração estão sob o namespace `Config`, mas não estão em `App\Config` como seria de se esperar. Isso permite que os arquivos principais do sistema sempre possam localizá-los, mesmo quando o namespace da aplicação for alterado.

Mapa de classe

O mapa de classe é utilizado de forma extensiva pelo CodeIgniter para obter o último desempenho do sistema, sem atingir as

chamadas adicionais `is_file()` do sistema de arquivos. Você pode usar o mapa de classe para vincular as bibliotecas de terceiros que não têm namespace:

```
$classmap = [
    'Markdown' => APPPATH .'third_party/markdown.php'
];
```

A chave de cada linha é o nome da classe que você deseja localizar, e o valor é o caminho o local onde o arquivo se encontra.

Suporte herdado

Se nenhum dos métodos anteriores encontrar a classe e ela não utilizar namespace, o autoloading procurará nos diretórios `/app/Libraries` e `/app/Models` para tentar localizar os arquivos. Isso fornece uma medida auxiliar para facilitar a transição de versões anteriores. Não existem configurações para esse tipo de suporte.

Suporte ao Composer

O suporte ao Composer é iniciado automaticamente por padrão. Ele procura o arquivo de autoloading do Composer em `ROOTPATH.'vendor/autoload.php'`. Caso você precise alterar o local desse arquivo por qualquer motivo, poderá modificar o valor definido em `/app/Config/Constants.php`.

```
defined('COMPOSER_PATH') || define('COMPOSER_PATH', ROOTPATH .
'vendor/autoload.php');
```

Se o mesmo namespace for definido no CodeIgniter e no Composer, o autoloading do CodeIgniter será o primeiro a ter a chance de localizar o arquivo.

Capítulo 13
Serviços

No CodeIgniter, todas as classes são fornecidas como *serviços*, o que significa que, em vez de codificar um nome de classe para carregar, as classes a serem chamadas são definidas em um arquivo de configuração bastante simples. Esse arquivo atua como um tipo de criador de novas instâncias da classe.

Imagine que na sua aplicação você precisa fazer uso da classe `Timer` durante um processo de debug. A forma mais simples e rápida de instanciar a classe é:

```
$timer = new \CodeIgniter\Debug\Timer();
```

Com o passar do tempo, sua aplicação vai sendo escalada e você precisa utilizar métodos de debug mais robustos envolvendo tempo para obter relatórios mais completos. Por padrão você terá que localizar todos os locais na sua aplicação onde a classe `Timer` foi utilizada, o que pode ser bastante demorado e aumentar as chances de erros na aplicação.

É nesse ponto que entram os serviços. Em vez de criar uma instância da classe manualmente, você deixa que uma classe central crie essa instância.

A classe contém apenas um método para cada classe que será utilizada como serviço, e esse método retorna uma instância compartilhada dessa classe, passando quaisquer dependências que existam para ela.

A chamada para a classe fazendo uso de serviços é a seguinte:

```
$timer = \Config\Services::timer();
```

Sempre que a implementação precisar ser alterada, você poderá modificar o arquivo de configuração dos serviços e a alteração

ocorrerá automaticamente por toda a aplicação.

> **ATENÇÃO**
>
> É recomendado criar serviços apenas nos controladores. Outros arquivos, como `models` e `libraries`, por exemplo, devem ter as dependências passadas para o construtor ou por meio de um método `setter`.

Funções de conveniência

Existem duas funções utilizadas na obtenção de serviços, e elas estão sempre disponíveis na aplicação. A primeira é `service()`, que retorna uma nova instância do serviço solicitado. Ela recebe como único parâmetro o nome do serviço.

```
$logger = service('logger');
```

Se o método de criação exigir parâmetros adicionais, eles poderão ser transmitidos após o nome do serviço:

```
$renderer = service('renderer',AOPPPATH.'views/');
```

A segunda função é `single_service()`. Ela funciona como `service`, mas retorna uma nova instância da classe:

```
$logger = single_service('logger');
```

13.1 Definindo serviços

Para que os serviços funcionem bem, você deve poder confiar em cada classe que possui uma API ou interface, e no CodeIgniter quase todas as classes fornecem uma interface à qual eles aderem.

Sempre que desejar estender ou substituir as classes principais, você só precisa garantir que atenda aos requisitos da interface e saber que as classes são compatíveis.

A classe `RouterCollection` implementa o `RouterCollectionInterface`. Quando você desejar criar uma substituição que ofereça uma forma diferente de criar rotas, basta criar uma nova classe que implemente o `RouterCollectionInterface`:

```
class MyRouter implements \CodeIgniter\Router\RouteCollectionInterface
{
    // Implemente os métodos da classe aqui
}
```

Feito isso, modifique `/app/Config/Services.php` para criar uma nova instância de `MyRouter` em vez de `CodeIgniter\Router\RouterCollection`:

```
public static function routes()
{
    return new \App\Router\MyRouter();
}
```

Permitindo parâmetros

Em algumas situações pode ser necessário passar parâmetros de configuração para a classe no momento de instanciá-la. Com os serviços, esse trabalho se torna bem simples.

Vamos utilizar como exemplo o serviço de renderização das views. Essa classe deve ser capaz de encontrar as views em `APPPATH.'views/'` e a implementação deve permitir ao desenvolvedor alterar esse caminho caso sua aplicação não utilize o diretório padrão do CodeIgniter.

Sendo assim, a classe aceita `$viewPath` como parâmetro construtor, deixando o método da seguinte forma:

```
public static function renderer($viewPath=APPPATH.'views/')
{
    return new \CodeIgniter\View\View($viewPath);
}
```

Sempre que for necessário alterar o caminho do diretório das views basta informá-lo na configuração do serviço:

```
$renderer = \Config\Services::renderer('/shared/views');
```

Classes compartilhadas

É possível criar várias instâncias de um mesmo serviço, mas em alguns casos pode ser necessário exigir que apenas uma única instância do serviço seja criada, e isso é facilmente implementado através do método `getSharedInstance()`.

Esse método verifica se uma instância está criada e salva dentro da classe e, se não, cria uma nova. Todos os métodos de criação fornecem um valor `$getShared = true` como último parâmetro, e você precisa segui-lo.

```
class Services
{
    public static function routes($getShared = false)
    {
        if (! $getShared)
        {
            return new \CodeIgniter\Router\RouteCollection();
        }

        return static::getSharedInstance('routes');
    }
}
```

Identificando serviços

No CodeIgniter, a identificação de serviços informados no arquivo `/app/Config/Services.php` ocorre de forma automática desde que você tenha definido um namespace. Para que os arquivos personalizados

dos serviços sejam identificados, eles devem atender a alguns requisitos:

- Seu namespace deve ser definido em `/app/Config/Autoload.php`
- Dentro do namespace, o arquivo deve ser encontrado em `/app/Config/Services.php`
- Ele deve estender `CodeIgniter\Config\BaseServices`

Você se lembra da aplicação da empresa `Abc` que abordei em capítulos anteriores? Ela possui um diretório `Blog` na raiz da aplicação, o que mantém o módulo do blog com controladores, models, views etc., e você precisa disponibilizar algumas das classes como serviço.

O primeiro passo é criar um arquivo `Blog/Config/Services.php` com a estrutura a seguir:

```
<?php namespace Blog\Config;

use CodeIgniter\Config\BaseService;

class Services extends BaseService
{
    public static function postManager()
    {
        // Implementações do método
    }
}
```

Quando você desejar obter o serviço de postagens de qualquer controlador, basta usar a classe `Config\Services` da estrutura para obter o serviço:

```
$postManager = Config\Services::postManager();
```

ATENÇÃO

Se vários arquivos de serviços tiverem o mesmo nome de método, o primeiro encontrado será a instância retornada.

Capítulo 14
Requisições HTTP

Durante o desenvolvimento de aplicações web, as requisições e respostas são feitas via HTTP, e para tirar o máximo de proveito disso no CodeIgniter é preciso compreender como essas requisições e respostas funcionam junto aos conceitos do HTTP.

14.1 Visão geral

O HTTP (*HyperText Transfer Protocol*) é uma convenção baseada em texto que permite a comunicação entre duas máquinas. Quando um navegador faz a requisição por uma página, ele pergunta ao servidor se pode obtê-la. Então o servidor prepara a página e envia uma resposta ao navegador solicitante.

Seu objetivo ao desenvolver aplicações web é entender as requisições que estão sendo feitas pelo cliente (navegador web, aplicativo para smartphones e tablets etc.), processá-las adequadamente no servidor e responder ao cliente.

A solicitação

Sempre que um cliente faz uma solicitação, uma mensagem é enviada para o servidor com informações necessárias para que ele possa compreender o que o cliente está querendo, processar a requisição e enviar a resposta:

```
GET / HTTP/1.1
Host livrocodeigniter.com.br
Accept: text/html
User-Agent: Chrome/46.0.2490.80
```

Essa mensagem exibe as informações necessárias para saber o que o cliente está solicitando. Ela informa o método para a solicitação (`GET` , `POST` , `DELETE` etc.) e a versão do protocolo HTTP compatível. Ela também inclui vários cabeçalhos de solicitação opcionais contendo uma variedade de informações úteis para que o servidor compreenda o que o cliente está querendo.

A resposta

Após receber a solicitação, a aplicação processa essas informações no servidor, gerando o conjunto de dados de saída para enviar ao cliente. Esse conjunto de informações também é representado através de uma mensagem de texto semelhante à exibida a seguir:

```
HTTP / 1.1 200 OK
Servidor: nginx / 1.8.0
Data: quinta-feira, 27 de fevereiro de 2020 07:48:22 GMT
Tipo de Conteúdo: text / html; charset = UTF-8

<html>
        <!-- Código HTML da resposta -->
</html>
```

A resposta informará ao cliente a versão do HTTP e a parte mais importante da resposta, que é o código de status da operação (200). Esse código de status é padronizado e possui um significado específico de modo a permitir que o cliente interprete devidamente a resposta, assim como o servidor interpretou a requisição.

Você pode ver uma lista completa dos status de respostas HTTP no link a seguir:

https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status

Trabalhando com solicitações e respostas

O PHP possui diversas maneiras de interagir com os cabeçalhos de solicitação e resposta. Já no CodeIgniter há uma abstração para

que seja possível trabalhar com uma interface mais enxuta e consistente.

A classe `IncomingRequest` é uma representação orientada a objeto da solicitação HTTP. Ela fornece tudo o que você precisa para trabalhar com solicitações e respostas.

```
use CodeIgniter\HTTP\IncomingRequest;

$request = new IncomingRequest(new \Config\App(), new
\CodeIgniter\HTTP\URI());

// a URI requisitada
$request->uri->getPath();

// Recupera as variáveis $_GET e $_POST
$request->getVar('foo');
$request->getGet('foo');
$request->getPost('foo');

// Recupera JSON
$request->getJSON();

// Recupera variáveis de servidor
$request->getServer('Host');

// Recupera o cabeçalho de uma solicitação HTTP (é case-sensitive)
$request->getHeader('host');
$request->getHeader('Content-Type');

// Recupera o método da solicitação
$request->getMethod();  // GET, POST, PUT, etc
```

Essa classe trabalha na maior parte do tempo em segundo plano. Os métodos `isAJAX()` e `isSecure()` verificam vários métodos diferentes para determinar a resposta correta.

O método `isAJAX()` depende do cabeçalho `X-Requested-With`, que em alguns casos não é enviado por padrão em requisições XHR feitas via JavaScript (i.e. fetch). Veja no final do capítulo como contornar esse problema.

O CodeIgniter também fornece uma classe `Response`, uma representação orientada a objetos para respostas HTTP, sendo uma forma simples e poderosa de construir respostas para o cliente:

```
use CodeIgniter\HTTP\Response;

$response = new Response();

$response->setStatusCode(Response::HTTP_OK);
$response->setBody($output);
$response->setHeader('Content-type', 'text/html');
$response->noCache();

// Envia o resultado da solicitação para o cliente
$response->send();
```

Além disso, a classe `Response` permite trabalhar a camada de cache HTTP para obter melhor desempenho.

Contornando o problema das requisições AJAX

O uso de JavaScript na parte do cliente está se tornando cada vez mais comum e consolidado, e em grande parte eles consomem dados através de APIs.

Essas APIs quando recebem uma solicitação geralmente verificam a origem, se foi uma solicitação direta ou se foi via XHR, e no caso de APIs desenvolvidas com CodeIgniter 4 é utilizado o método `incominRequest::isAJAX()` para poder verificar a origem da solicitação.

Porém, o método `isAJAX()` usa o cabeçalho `X-Requested-With` para saber de onde veio a solicitação, se ela é XHR ou não. As

implementações mais recentes do JavaScript (i.e. fetch) não estão mais enviando esse cabeçalho e com isso o método `isAJAX()` não funciona corretamente.

Para resolver esse problema e poder continuar utilizando o método `isAJAX()`, é necessário fazer o envio manual (ou forçado, como alguns gostam de chamar) do cabeçalho `X-Requested-With`.

Veja a seguir como fazer esse envio forçado em JavaScript puro, jQuery, VueJS e React.

JavaScript Puro (Fetch API)

```
fetch(url, {
        method: “get”, headers: {
                “Content-Type”: “application/json”,
                “X-Requested-With”: “XMLHttpRequest”
        }
});
```

jQuery

```
        $.ajax({
                url: “your url”,
                headers: {‘ X-Requested-With’ : ‘ XMLHttpRequest’ }
        });
```

VueJS *usando AXIOS

```
    axios.defaults.headers.common[‘ X-Requested-With’ ] = ‘ XMLHttpRequest’ ;
```

React (usando AXIOS)

```
    axios.get(“your url”, {
            headers: {
                    ‘ Content-Type’ : ‘ application/json’
                }
        })
```

Capítulo 15
URLs

As URLs no CodeIgniter são projetadas para serem mecanismos de pesquisa e totalmente *user-friendly*. Em vez de utilizar uma estrutura de URL composta por variáveis e strings de consulta, são utilizados segmentos:

```
meusite.com.br/blog/busca
```

15.1 Segmentos da URI

Os segmentos da URI representam de alguma forma a abordagem MVC, facilitando a identificação da estrutura de processamento por parte do desenvolvedor. No exemplo a seguir, a composição é a seguinte:

- o primeiro segmento, `/blog` , representa o controlador;
- o segundo segmento, `/busca` , representa o método a ser executado.

```
//host/classe/método
meusite.com.br/blog/busca
```

Mas também é possível ter URLs com parâmetros, como a apresentada a seguir:

```
//host/classe/método/ID
meusite.com.br/blog/busca/codeigniter
```

- o primeiro segmento, `/blog` , representa o controlador;
- o segundo segmento, `/busca` , representa o método a ser executado;

- o terceiro segmento, `/codeigniter`, representa o parâmetro a ser passado para o método.

O CodeIgniter disponibiliza a biblioteca `URI` e o helper `URL` que auxiliam no trabalho com URLs. Ele também permite remapear as URLs utilizando o recurso de roteamento de URI, o que dá maior flexibilidade dentro da aplicação, uma vez que você pode estruturar as URLs sem a necessidade de seguir os modelos apresentados anteriormente.

Por exemplo, a URL para a leitura de um post no blog poderia ser `meusite.com.br/blog/titulo-do-post`. O controlador está explícito na URL, é o `Blog`, mas o método não está, pois no lugar dele é passado o título do post em formato *user-friendly*. Para que essa URL chame o método correto no controlador `Blog` é necessário definir uma rota capaz de apontar para o método correto no controlador, passando o segundo segmento da URL como um parâmetro.

```
$routes->get('/blog/(:any)', 'Blog::lerPost/$1');
```

Mais adiante neste livro você verá um capítulo dedicado exclusivamente a falar sobre rotas.

15.2 Removendo o arquivo index.php da URL

Aplicações desenvolvidas com CodeIgniter têm como padrão o uso de `index.php` logo após o nome do `host`, o que quebra o conceito apresentado anteriormente. Mas é possível remover o `index.php` da URL desde que o servidor onde a aplicação estiver hospedada tenha suporte à reescrita de URLs.

```
meusite.com.br/index.php/blog/titulo-do-post
```

A forma de reescrita de URLs varia conforme o tipo de servidor utilizado, sendo assim, a seguir você verá a forma de remoção para

os servidores web mais comuns.

Apache

Para que a reescrita de URLS funcione no servidor web Apache é preciso que a extensão `mod_rewrite` esteja instalada e ativada. Com a extensão devidamente ativada, você deverá utilizar um arquivo `.htaccess` no diretório `/public` da aplicação contendo regras simples que fazem a remoção sem afetar o funcionamento da aplicação.

No exemplo a seguir, o arquivo tratará qualquer solicitação HTTP que não seja de diretórios e arquivos existentes como uma solicitação ao arquivo `index.php` . Dessa forma, todas as URLs da aplicação responderão sem o `index.php` explícito nelas.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1 [L]
```

> **ATENÇÃO**
>
> Dependendo das configurações do servidor web, essa regra pode não funcionar. Então é importante ficar atento às configurações do servidor em caso de erro.

NGINX

No NGINX você utiliza um bloco de localização e a diretiva `try_files` para realizar a remoção do `index.php` .

Primeiro, ele vai procurar por diretório ou arquivo correspondente à URI que está sendo acessada, e se não encontrar envia a solicitação ao arquivo `index.php` .

```
location / {
    try_files $uri $uri/ /index.php/$args;
}
```

Capítulo 16
Helpers

Os *helpers* são conjuntos de funções para otimizar o processo de escrita de código, tornando mais fluido e enxuto.

Eles funcionam como auxiliares na execução de tarefas dentro da aplicação. Cada helper é uma coleção de funções organizadas de forma específica dentro da aplicação. O CodeIgniter traz cerca de 12 helpers nativos, um volume menor que o CodeIgniter 3, que possui 21.

Helpers nativos

- Array
- Cookie
- Date
- File System
- Form
- HTML
- Inflector
- Number
- Security
- Text
- URL
- XML

Diferente dos demais arquivos que compõem uma aplicação desenvolvida com CodeIgniter 4, os helpers não utilizam o formato de Orientação a Objetos. São funções simples, cada uma executa uma tarefa específica, sem depender de outras.

Os helpers não são carregados junto da aplicação, é preciso carregá-los conforme a necessidade. Após carregá-los ficam disponíveis de forma global nos controladores e nas views.

Os helpers nativos do CodeIgniter ficam armazenados em `system/Helpers` (no caso de instalação manual através de arquivo ZIP ou via Composer com `codeigniter4/framework`) ou em `vendor/codeigniter4/framework/system/Helpers` (para instalação via Composer usando `codeigniter4/appstarter`) e os helpers da aplicação, que podem ser construídos conforme a necessidade, ficam em `/app/Helpers`.

Quando você solicitar o carregamento de um helper na aplicação, será verificada a existência dele primeiro no diretório `/system/Helpers` ou `vendor/codeigniter4/framework/system/Helpers` e depois no `/app/Helpers`.

16.1 Carregando um helper

Para carregar um helper no CodeIgniter 4, basta utilizar o método `helper('nome-do-helper');`. Por exemplo, se quiser carregar o helper `Form`, basta chamar `helper('form');`.

Caso queira carregar múltiplos helpers, você pode utilizar um array para informar a lista de helpers a serem carregados:

```
helper(['form', 'cookie']);
```

Um helper pode ser carregado dentro de métodos do controlador, ou mesmo nas views (isso não é uma boa prática, mas é possível). Você pode carregá-los dentro do método construtor dos controladores para que eles estejam disponíveis automaticamente em todos os métodos do controlador.

> **INFORMAÇÃO**
>
> O helper `URL` é sempre carregado pela aplicação, não sendo necessário o seu carregamento manual.

Os helpers também podem ser carregados de fora dos diretórios padrões do CodeIgniter, desde que o caminho deles possa ser identificado através de namespace devidamente configurado no arquivo `/app/Configs/Autoloader.php`.

Voltando ao exemplo de módulo `Blog` que foi apresentado em capítulos anteriores, vamos supor que esse módulo tenha helpers específicos. Sendo assim, os helpers devem ficar em `/Blog/Helpers` e serem nomeados, por exemplo, `blog_helper.php`. Dessa forma é possível carregar um helper de um módulo utilizando o namespace do mesmo, que seria `helper('\Blog\blog');`.

> **ATENÇÃO**
>
> A definição de namespace para os helpers se dá apenas no formato da estrutura de arquivos para facilitar a localização e o carregamento deles. A estrutura do arquivo do helper continua sendo composta apenas pelas funções, sem orientação a objeto e sem namespace definido de forma explícita no arquivo.

16.2 Usando um helper

Após carregar o helper que contém a função que necessita utilizar, você a chamará como uma função padrão do PHP. Suponhamos que você em algum momento queira obter a URL atual do usuário da sua aplicação. Para isso você deverá carregar o helper `URL` e em seguida chamar a função `current_url()`.

```
helper('url');
$current_url = current_url();
```

Estendendo um helper

Os helpers podem ser estendidos assim como outras partes do framework, para isso basta criar um arquivo no diretório `/app/Helpers` com o mesmo nome do arquivo padrão do framework e inserir as funções adicionais de que sua aplicação necessita.

Também é possível substituir funções padrões, adequando-as ao funcionamento da aplicação, bastando criar uma função com o mesmo nome e a nova rotina dentro do arquivo do helper em `/app/Helpers`.

Veja a seguir a ordem de carregamento dos helpers:

- `app/Helpers`
- `{namespaces}/Helpers`
- `system/Helpers` ou `vendor/codeigniter4/framework/system/Helpers`

> **INFORMAÇÃO**
>
> Os helpers são procedurais e não podem ser estendidos no sentido programático tradicional. O termo "estender" aqui é utilizado para facilitar a compreensão.

16.3 Visão geral dos principais helpers nativos

A seguir você encontrará uma visão geral dos principais helpers nativos do CodeIgniter 4 e para cada helper descrito neste capítulo você encontrará um exemplo com código comentado no repositório do livro no GitHub (https://github.com/jlamim/livro-codeigniter4).

Cookie Helper

Esse helper fornece várias funções capazes de simplificar o trabalho com `cookies`. Sua finalidade é parecida com a da biblioteca *Session*,

porém ele atua exclusivamente com `cookies`. Algumas das funções disponíveis no helper:

- `set_cookie()`: utilizado para criar um cookie no browser do usuário.
- `get_cookie()`: utilizado para recuperar um cookie criado com o método `set_cookie()`.
- `delete_cookie()`: utilizado para remover cookies criados com o método `set_cookie()`.

FileSystem Helper

Esse helper é muito útil quando a aplicação precisa lidar com sistemas de arquivos, pois ele fornece funções para fazer mapeamento de diretórios, criar arquivos, recuperar informações sobre arquivos e diretórios, entre outras. Algumas das funções disponíveis no helper:

- `directory_map()`: utilizada para realizar o mapeamento de um diretório, retornando um `array` com o conteúdo.
- `write_file()`: utilizada para criar arquivos.
- `delete_files()`: utilizada para remover arquivos.
- `get_filenames()`: utilizada para recuperar o nome dos arquivos em um determinado diretório.

Form Helper

Muitos programadores utilizam esse helper para substituir a escrita de código HTML utilizado na criação de formulários. Algumas das funções disponíveis no helper:

- `form_open()`: imprime na tela a tag `<form>` com os atributos devidamente especificados como parâmetros da função.
- `form_open_multipart()`: imprime na tela a tag `<form>` com os atributos devidamente especificados como parâmetros da função e o atributo `multipart`.

- `form_input()` : imprime na tela um campo `<input>` com os atributos devidamente especificados como parâmetros da função.
- `form_textarea()` : imprime na tela um campo `<textarea>` com os atributos devidamente especificados como parâmetros da função.

HTML Helper

Nesse helper você terá disponíveis diversas funções capazes de substituir a escrita de elementos HTML no código. Algumas das funções disponíveis no helper:

- `img()` : imprime na tela a tag `<img>` com os atributos devidamente especificados como parâmetros da função.
- `link_tag()` : imprime na tela a tag `<a>` com os atributos devidamente especificados como parâmetros da função.
- `ul()` : imprime na tela o conjunto `<ul><li></li></ul>` com os atributos e valores especificados como parâmetros na função.
- `video()` : imprime na tela a tag `<video>` com os atributos devidamente especificados como parâmetros da função.

Number Helper

Esse helper traz funções par auxiliar o trabalho com dados numéricos de forma compatível com o `timezone` da aplicação. Algumas das funções disponíveis no helper:

- `number_to_size()` : formata números como bytes adicionando o sufixo apropriado.
- `number_to_amount()` : formata números de forma legível por humanos até a casa dos quatrilhões.
- `number_to_currency()` : formata números no formato de moeda, como USD, BRL e EUR, por exemplo.

- `number_to_roman()` : formata números para o modo romano de numeração.

Security Helper

O uso desse helper deveria ser obrigatório em todas as aplicações, pois ele traz funções que ajudam a tornar a aplicação mais segura. Algumas das funções disponíveis no helper:

- `strip_image_tags()` : retira as tags `<img>` de uma string, deixando a URL da imagem como um texto simples.
- `encode_php_tags()` : converte tags PHP em entidades.

Text Helper

Um helper que auxilia no trabalho com textos permitindo a geração de texto randômico, limitação da quantidade de palavras e caracteres, conversão de caracteres acentuados, entre outras funcionalidades. Algumas das funções disponíveis no helper:

- `random_string()` : gera uma string com caracteres randômicos.
- `word_limiter()` : limita a quantidade de palavras em um texto.
- `character_limiter()` : limita a quantidade de caracteres em um texto.
- `convert_accented_characters()` : converte os caracteres acentuados em caracteres se acento.
- `highlight_code()` : adiciona uma marcação colorida a blocos de código inseridos com a tag `<code>` .

URL Helper

Outro helper muito utilizado pelos programadores pois ele otimiza e padroniza o trabalho com URLs. Algumas das funções disponíveis no helper:

- `site_url()` : retorna a URL da aplicação.
- `base_url()` : retorna a URL base da aplicação conforme configurada em `/app/Configs/App.php` ou no arquivo `env` .

- `current_url()` : retorna a URL da página que o usuário está acessando.
- `prep_url()` : adiciona o protocolo `http://` a um link.

Capítulo 17
Bibliotecas

O CodeIgniter traz cerca de 15 bibliotecas nativas, um volume bem menor que o CodeIgniter 3, que possui 29 bibliotecas nativas. Essa diferença se dá por conta da reformulação de toda a estrutura do framework. Muitas das bibliotecas passaram a integrar outras classes e recursos e algumas foram removidas pois não eram fundamentais para o funcionamento do framework, pouco utilizadas por programadores e praticamente faziam apenas volume na estrutura de arquivos.

A seguir você encontrará uma visão geral das principais bibliotecas nativas do CodeIgniter 4 e para cada biblioteca descrita neste capítulo você encontrará um exemplo com código comentado no repositório do livro no GitHub (https://github.com/jlamim/livro-codeigniter4).

Mas você também pode criar suas próprias bibliotecas ou estender as bibliotecas já existentes para que obtenha os recursos necessários à sua aplicação.

17.1 Caching Driver

A biblioteca *Caching Driver* traz uma das formas de armazenamento em cache mais rápidas e dinâmicas.

Alguns dos drivers de cache suportados pela biblioteca são: *Memcached*, *WinCache*, *Redis* e *Dummy*. Todos esses drivers exigirão requisitos específicos do servidor, e caso esses requisitos não sejam atendidos uma exceção fatal será lançada pela aplicação.

As configurações para o devido funcionamento dessa biblioteca são feitas em `/app/Config/Cache.php`, onde você implementa todas as definições e no código chama apenas os métodos com seus devidos parâmetros. Por exemplo, o método `save($key, $data[, $ttl = 60[, $raw = FALSE]])` recebe como parâmetros a chave de identificação do cache, os dados a serem armazenados, o tempo em que esses dados devem permanecer no cache, a informação se o valor bruto deve ser armazenado, sendo este último parâmetro muito utilizado quando o driver escolhido é o *Memcached*.

Outro método muito utilizado é o `get($key)`, que recebe como parâmetro o nome que identifica o cache armazenado e o retorna para a aplicação.

Alguns outros métodos disponíveis na biblioteca:

- `isSupported()`: verifica se há suporte para uso do serviço de cache.
- `delete($key)`: remove o cache de acordo com a chave especificada no parâmetro `$key`.
- `clean()`: limpa todo o cache.

17.2 CURLRequest Class

A biblioteca *CURLRequest Class* é um cliente HTTP baseado em CURL que permite comunicação com outros sites e servidores na internet. Através dela é possível obter informações e conteúdos de outros sites e consumir APIs por exemplo.

Ela foi modelada a partir de uma das bibliotecas HTTP mais famosas, a *Guzzle HTTP Client*, de modo que, caso sua aplicação necessite de mais funcionalidades para requisições HTTP do que a *CURLRequest* possui, você pode migrar para a *Guzzle* tendo que fazer poucas alterações no código da sua aplicação.

Para que essa biblioteca funciona corretamente é necessário que a biblioteca *cURL* do PHP esteja instalada, caso contrário a aplicação retornará um erro.

Alguns dos métodos disponíveis na biblioteca:

- `request($method, $url, $options)` : responsável por executar uma requisição à URL definida no parâmetro `$url` utilizando um dos métodos HTTP definido no parâmetro `$method` e com opções definidas em `$options` .
- `get($url, $options)` : executa uma requisição `GET` à URL definida no parâmetro `$url` e com as opções definidas em `$options` .
- `delete($url, $options)` : executa uma requisição `DELETE` à URL definida no parâmetro `$url` e com as opções definidas em `$options` .
- `post($url, $options)` : executa uma requisição `POST` à URL definida no parâmetro `$url` e com as opções definidas em `$options` .

Algo bastante interessante dessa biblioteca é que você pode trabalhar com ela de forma integrada à `CodeIgniter\HTTP\Response` , o que lhe permitirá construir respostas HTTP de forma mais objetiva e rápida para a sua aplicação.

17.3 Email Class

A biblioteca *Email Class* é responsável por auxiliar no envio de emails da sua aplicação e não sofreu muitas mudanças em relação à existente no CodeIgniter 3. Suas configurações continuam sendo definidas no arquivo de configuração, que agora fica em `/app/Config/Email.php` e os métodos permaneceram os mesmos.

Alguns dos métodos disponíveis na biblioteca:

- `setFrom($from, $name, $returnPath)` : define o email (`$from`) e o nome (`$name`) de quem está enviando o email, além do email que deverá receber a resposta (`$returnPath`).
- `setFrom($to)` : define o email (`$to`) de quem receberá o email.
- `setCC($to)` : define o email (`$co`) de quem receberá uma cópia do email.
- `setSubject($subject)` : define o asunto (`$subject`) do email.
- `setMessage($message)` : define a mensagem (`$message`) do email.
- `send($autoClear)` : faz o envio do email, e se `$autoClear` for definida como `TRUE` então limpa todos os parâmetros após o envio.

17.4 Encryption Service

A biblioteca *Encryption Service* é capaz de gerar chaves secretas com criptografia de dados simétrica bidirecional.

Até o momento em que este livro estava sendo escrito ela suportava apenas *OpenSSL*, já que a partir do PHP 7.2 a extensão `MCrypt` foi descartada. Sendo assim, ela não é uma solução completa de criptografia, mas ajuda em situações onde o nível de criptografia que se necessita é baixo.

Uma recomendação importante da própria equipe de desenvolvedores do CodeIgniter é que essa biblioteca não seja utilizada para a geração e armazenamento de senhas, sugerindo inclusive o uso da extensão nativa do PHP, a *Password Hashing*.

> **DOCUMENTAÇÃO OFICIAL DA PASSWORD HASHING**
>
> http://php.net/password

Alguns dos métodos disponíveis na biblioteca:

- `createKey($length)` : cria uma chave secreta randômica com o tamanho passado no parâmetro `$length` .
- `encrypt($data, $params)` : faz a encriptação de dados (`$data`) de acordo com os parâmetros enviados em `$params` .
- `decrypt($data, $params)` : faz a decriptação de dados (`$data`) de acordo com os parâmetros enviados em `$params` .

17.5 Image Manipulation Class

A biblioteca *Image Manipulation Class* permite manipular imagens dentro da aplicação mudando seu tamanho, criando thumbnails, rotacionando e cortando imagens, e também permite adicionar marca d'água nas imagens. Ela suporta as extensões PHP `GD/GD2` e `ImageMagick` .

Assim como a biblioteca *Email Class*, essa também não sofreu muitas modificações em relação à existente no CodeIgniter 3.

Alguns dos métodos disponíveis na biblioteca:

- `convert($imageType)` : ao salvar a imagem faz a conversão para o formato passado no parâmetro `$imageType` .
- `fit($width, $height, $position)` : corta a imagem de acordo com a largura (`$width`) e altura (`$height`) informados e o posicionamento do corte (`$position`).
- `flip($dir)` : faz o giro da imagem na horizontal ou vertical, conforme o parâmetro `$dir` .
- `rotate($angle)` : faz a rotação da imagem no ângulo informado em `$angle` .

17.6 Pagination

A biblioteca *Pagination* disponibiliza de uma maneira simples e flexível uma estrutura de links de paginação para registros. Ela trabalha em conjunto com o `Model` e suporta múltiplas paginações na mesma `view`.

Por ser configurável, você pode utilizá-la em qualquer aplicação independente do layout, se está utilizando bibliotecas CSS como `Bootstrap` ou `Material Design`.

Alguns dos métodos disponíveis na biblioteca:

- `links()` : retorna a estrutura de paginação com os devidos links.
- `getFirst()` : retorna o link da primeira página.
- `getLast()` : retorna o link da última página.
- `getPrevious()` : retorna o link da página anterior.
- `getNext()` : retorna o link da próxima página.

17.7 Security Class

A biblioteca *Security Class* contém métodos para ajudar na proteção da aplicação contra-ataques do tipo CSRF (Cross-Site Request Forgery). Ela permite adicionar filtros de segurança em formulários e também URIs utilizadas como endpoints de APIs, combinando expressões regulares às URIs.

Também é possível utilizá-la junto do helper `Form` para inserir campos ocultos nos formulários para execução das validações de segurança através dos métodos `csrf_token()` e `csrf_hash()`, ou `csrf_field()`.

Se for feita uma requisição com dados em formato JSON, algo muito comum em APIs, o token CSRF pode ser passado como parâmetro. Uma outra forma de enviar o token CSRF é no cabeçalho HTTP da requisição, e isso pode ser feito utilizando o método `csrf_header()`.

17.8 Session Library

A biblioteca *Session Library* permite armazenar informações da aplicação enquanto o usuário a está utilizando, por exemplo, informações de autenticação para verificação do status do usuário, páginas para redirecionamento em caso de o usuário tentar acessar uma página restrita sem estar logado. Ela já existia no CodeIgniter 3 e sofreu poucas mudanças na versão 4 do framework.

Os drivers suportados são: *Redis* e *Memcached*, além de ser possível armazenar em arquivos ou no próprio banco de dados.

Alguns dos métodos disponíveis na biblioteca:

- `destroy()` : destrói todos os dados de sessão.
- `get($key)` : retorna os dados da sessão armazenados para a chave definida em `$key` .
- `set($data)` : armazena os dados (`$data`) na sessão.
- `push($key,$data)` : adiciona mais dados (`$data`) a uma chave específica (`$key`) da sessão.

17.9 Throttler

A biblioteca *Throttler* traz uma forma muito simples de limitar a atividade a ser executada a um determinado número de tentativas em um intervalo de tempo predefinido. Esse é um recurso muito utilizado por APIs e aplicações que precisam limitar o número de realizações de uma mesma tarefa dentro de um espaço de tempo. Ela é uma versão simplificada do algoritmo *Token Bucket*. Você pode utilizá-la dentro da aplicação para qualquer coisa que necessite de controle de ações dentro de um intervalo de tempo predefinido, como tentativas de login com erro, envios de formulário.

Para que a biblioteca *Throttler* funcione corretamente é preciso que a biblioteca *Cache* esteja configurada e para um melhor desempenho é recomendado o uso de *Redis* ou *Memcached* para armazenamento desse cache.

Alguns dos métodos disponíveis na biblioteca:

- `check($key, $capacity, $seconds)` : verifica se o bucket (`$key`) teve o limite de execução (`$capacity`) excedido para o tempo especificado (`$seconds`).
- `getTokentime()` : retorna o número de segundos até que a execução possa ser realizada novamente.

17.10 Dates e Times

Essa é uma biblioteca nova, construída sob o objeto `DateTime` do PHP, com recursos da extensão `Intl` para converter data e hora entre fusos horários, retornando os valores para diferentes regiões, conforme os `timezones` do próprio PHP.

Alguns dos métodos disponíveis na biblioteca:

- `today()` : retorna a data atual.
- `yesterday()` : retorna a data do dia anterior.
- `tomorrow()` : retorna a data do próximo dia.
- `humanize()` : retorna uma `string` com a diferença entre duas datas, sendo uma delas a data atual de forma legível, por exemplo "1 ano atrás" em vez de "365".

Capítulo 18
Logs

Arquivos de log são de grande utilidade em aplicações web e o CodeIgniter permite trabalhar com logs de forma simples e eficiente.

Para registrar um log em arquivos locais, você pode utilizar o método `log_message()`, informando o nível do log no primeiro parâmetro e a mensagem no segundo parâmetro.

```
log_message('error','Não foi possível obter todas as informações.
Registros armazenados de forma incompleta.');
```

O CodeIgniter opera com 8 níveis de log, correspondentes aos níveis da RFC 5424 (https://tools.ietf.org/html/rfc5424):

- **debug:** informações detalhadas sobre depuração;
- **info:** informações importantes que a aplicação precise armazenar;
- **notice:** eventos normais, mas significativos para a aplicação;
- **warning:** ocorrências que exigem atenção, mas não são erros;
- **error:** erros em tempo de execução que não necessitem de ação imediata mas devem ser registrados e monitorados;
- **critical:** condições críticas da aplicação ou exceções não esperadas;
- **alert:** questões da aplicação com necessidade de ações imediatas;
- **emergency:** o sistema está inutilizado.

Para eventos de nível mais crítico o framework os registra de forma automática durante a execução da aplicação.

ATENÇÃO

Através do sistema de log não é possível alertar os administradores sobre os eventos, pois eles apenas registram as informações.

18.1 Configuração

No arquivo `/app/Configs/Logger.php` é possível modificar quais os níveis de registro serão utilizados assim como atribuir registradores para lidar com diferentes níveis.

A variável `$threshold` determina o nível de log a ser registrado, fazendo com que o sistema grave no arquivo de log apenas as notificações do nível informado.

É possível informar vários níveis, para isso basta utilizar um array com os valores que representam cada nível.

- 0 = Disables logging
- 1 = Emergency Messages
- 2 = Alert Messages
- 3 = Critical Messages
- 4 = Runtime Errors
- 5 = Warnings
- 6 = Notices
- 7 = Info
- 8 = Debug
- 9 = All Messages

```
public $threshold = array(2,3,4);
```

Usando vários manipuladores de log

O CodeIgniter suporta vários métodos de manipulação de log sendo executados ao mesmo tempo. Cada um desses manipuladores pode ser configurado para lidar com níveis específicos de log, ignorando os demais.

Por padrão, o framework traz 2 manipuladores:

O **manipulador de arquivos** *(é o manipulador recomendado)*, que é o padrão e criará um único arquivo de log armazenado no sistema de arquivos da aplicação (`/writable/logs`). E o **manipulador ChromeLogger** que está diretamente ligado à extensão *ChromeLogger* do navegador Google Chrome, e permite que as mensagens de log sejam exibidas na janela do console no navegador.

Esses manipuladores são configurados no arquivo de configuração `/app/Configs/Logger.php` , na propriedade `$handlers` , que nada mais é do que um array de manipuladores e suas devidas configurações.

Cada manipulador é especificado com a chave sendo o nome da classe com o namespace completo. O valor será um array com as propriedades específicas para cada manipulador. Nesse array haverá uma chave em comum chamada `handles` que conterá os níveis cujo manipulador deve atuar.

```
public $handlers = [
    'CodeIgniter\Log\Handlers\FileHandler' => [

            'handles' => ['critical', 'alert', 'emergency', 'debug',
'error', 'info', 'notice', 'warning'],
    ]
];
```

18.2 Modificando as mensagens

É comum alterar as mensagens de log com base no contexto do evento que está sendo registrado. Essa alteração é possível utilizando marcações entre chaves que correspondem às informações a serem inseridas nas mensagens.

No primeiro parâmetro temos o nível do log, no segundo, a mensagem contendo as marcações personalizadas e no terceiro, temos um array com as correspondências entre as marcações personalizadas e seus devidos valores.

Suponha que a empresa ABC quer manter um log de acessos ao seu blog e sempre que um usuário acessá-lo ela armazena o ID do usuário conectado e o endereço IP desse usuário. A estrutura de código para armazenamento desse log seria a seguinte:

```
$info = [
        'id' => $user->id,
        'ip_address' => $this->request->ip_address()
];

log_message('info', 'User {id} logged into the system from {ip_address}', $info);
```

Além desse log a empresa quer registrar um log para eventuais problemas que venham a ocorrer durante o processamento das requisições para assim melhorar a plataforma. Então os registros podem ser feitos utilizando um `try/catch` e recuperando os detalhes da `exception` para armazenar no log.

```
try
{
    //Código a ser executado pela aplicação
    //Se der erro nessa execução, chama o catch registra o log na sequência
}
catch (\Exception $e)
{
    log_message('error', '[ERROR] {exception}', ['exception' => $e]);
}
```

Você pode definir quantas marcações personalizadas forem necessárias para ter arquivos de log mais legíveis e compreensíveis e o CodeIgniter ajuda nesse processo disponibilizando um conjunto de marcações padronizadas que podem ser recuperadas na execução em questão:

- `{post_vars}` : variável `$_POST`
- `{get_vars}` : variável `$_GET`
- `{session_vars}` : variável `$_SESSION`
- `{env}` : nome do ambiente atual em que a aplicação está rodando
- `{file}` : nome do arquivo que está chamando a criação do log
- `{line}` : linha no arquivo `{file}` onde o a criação do log foi chamada
- `{env: variavel}` : o valor da variável em `$_ENV`

18.3 Usando registradores de terceiros

Como já foi falado, você pode utilizar outro registrador de log no CodeIgniter desde que ele se estenda de `Psr\Log\LoggerInterface` e seja compatível com a PSR3 (Logger Interface).

É importante garantir que o registrador de log utilizado seja identificado pelo sistema, ou seja, você deve adicioná-lo ao arquivo de configuração `/app/Config/Autoload.php` ou então através do *Composer*. Após essa configuração você precisa modificar `/app/Config/Services.php` para apontar o alias do registrador de log para o seu novo nome de classe.

A partir desse momento, qualquer chamada a `log_message()` utilizará a biblioteca configurada.

Capítulo 19
Tratamento de erros

O CodeIgniter é capaz de criar relatórios de erros através de Exceptions, tanto na *collection SPL* (uma coleção de interfaces e classes que servem para resolver problemas padrões na aplicação) quanto em *exceptions* personalizadas fornecidas pelo framework.

Dependendo da configuração do ambiente, a ação padrão quando um erro ou exception é lançada é exibir um relatório detalhado, a menos que a aplicação esteja sendo executada no ambiente de produção. Nesse caso uma mensagem mais genérica será exibida com o intuito de manter uma melhor experiência para o usuário.

Tela de erro em ambiente de produção

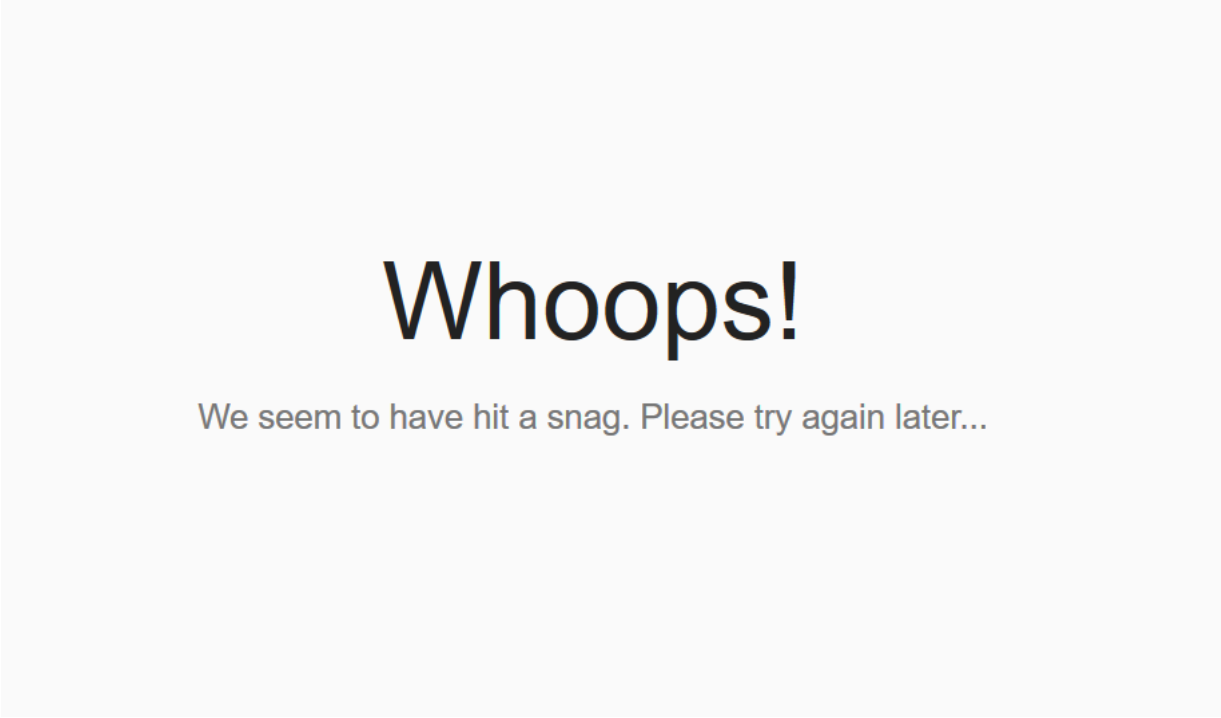


Figura 19.1: Tela de erro em ambiente de produção

Tela de erro em ambiente de desenvolvimento

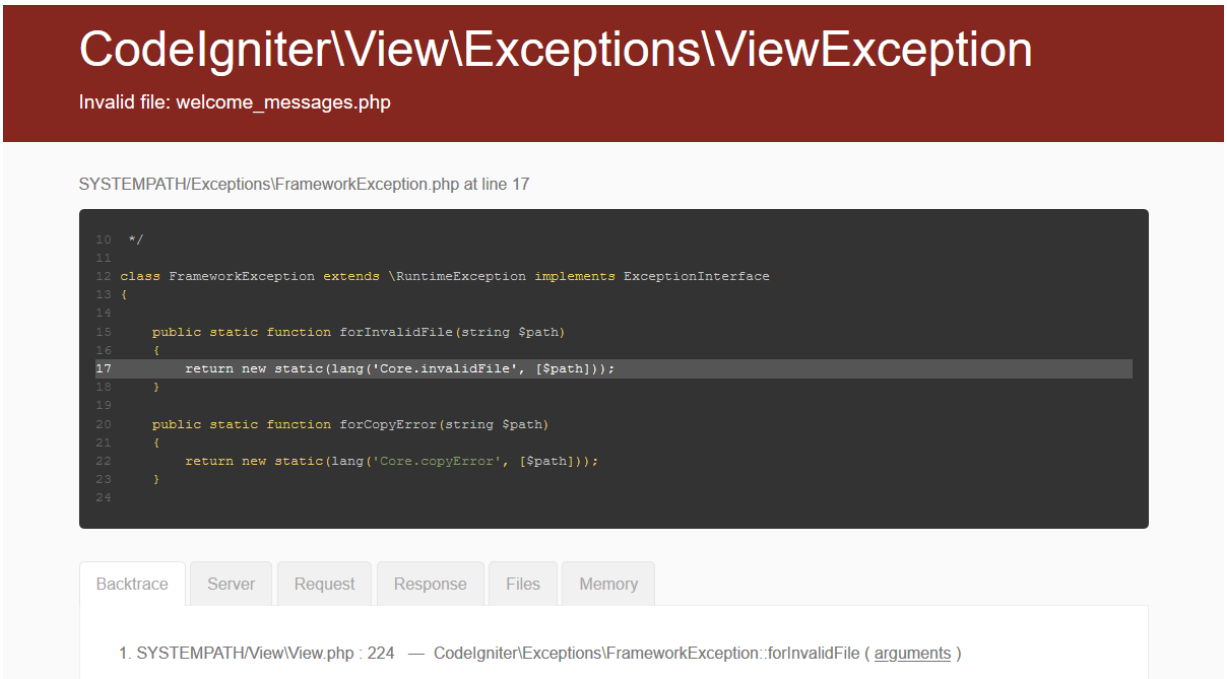


Figura 19.2: Tela de erro em ambiente de desenvolvimento

19.1 Usando exceptions

Exceções são eventos que acontecem quando o fluxo atual do script é interrompido e a execução é enviada a um manipulador de erros capaz de exibir a página de erro apropriada.

```
throw new \Exception("Mensagem relacionada à exceção que foi lançada");
```

Caso você esteja chamando um método que pode gerar uma exception, poderá capturá-lo usando o bloco `try/catch` . Se `$postModel` lançar uma exceção ela será capturada e o código contido no bloco `catch` será executado. Nesse exemplo, a execução que ocorria em `try` é encerrada e a mensagem de erro definida em `PostModel` é exibida.

```
try {
    $post = $postModel->find($id);
}
```

```
catch (\Exception $e)
{
    die($e->getMessage());
}
```

No exemplo anterior é capturado qualquer tipo de exception, porém, se quiser capturar tipos específicos de exception você pode, basta informar no parâmetro `catch` qual o tipo de exception deseja capturar.

```
try{
    //Código a ser executado
}
catch (\CodeIgniter\UnknownFileException $e)
{
    //Código a ser executado quando a exception for lançada, em caso de try falhar
}
```

19.2 Configuração

É um padrão do CodeIgniter exibir todos os erros nos ambientes de teste e desenvolvimento, e no ambiente de produção não exibir erros. Você deve se lembrar de que desativar os relatórios de erros (como acontece no ambiente de produção) não impede que os logs sejam gravados quando os erros acontecerem.

O arquivo de logs continua sendo gerado de acordo com as configurações aplicadas a ele, como foi visto no capítulo anterior.

Logging Exceptions

Todas as exceptions, com exceção das exceptions 404 (página não encontrada), são registradas, e isso pode ser ativado ou desativado definindo o valor de `$log` em `/app/Config/Exceptions.php` .

Para ignorar a execução em outros códigos de status além do 404, basta definir o código do status a ser ignorado em `$ignoredCodes`.

```
<?php namespace Config;

class Exceptions
{
    public $log = true;
    public $ignoredCodes = [ 404, 403, 402 ];
    public $errorViewPath = APPPATH . 'Views/errors';
}
```

> **ATENÇÃO**
>
> Caso as configurações de log não estejam definidas para registrar erros críticos, nos quais todas as exceptions são registradas, é possível que os registros das exceptions não ocorram.

19.3 Exceptions customizadas

O CodeIgniter permite personalizar exceptions e disponibiliza as seguintes exceptions personalizadas: `PageNotFoundException`, `ConfigException` e `DatabaseException`.

PageNotFoundException

É o tipo de exception utilizada para apontar erros 404, conhecidos como *Page Not Found*. Quando essa exception for lançada o sistema vai mostrar a página com a estrutura da view `/app/Views/errors/html/error_404.php`.

```
if (!$page = $postModel->find($id))
{
    throw \CodeIgniter\Exceptions\PageNotFoundException::forPageNotFound
```

```
();
}
```

Você pode personalizar todas as views de erros da aplicação tanto a nível de layout quanto a nível de rota. Se você tem uma página específica a ser exibida quando um erro 404 ocorrer, então ela deve ser informada no arquivo `/app/Config/Routes.php` .

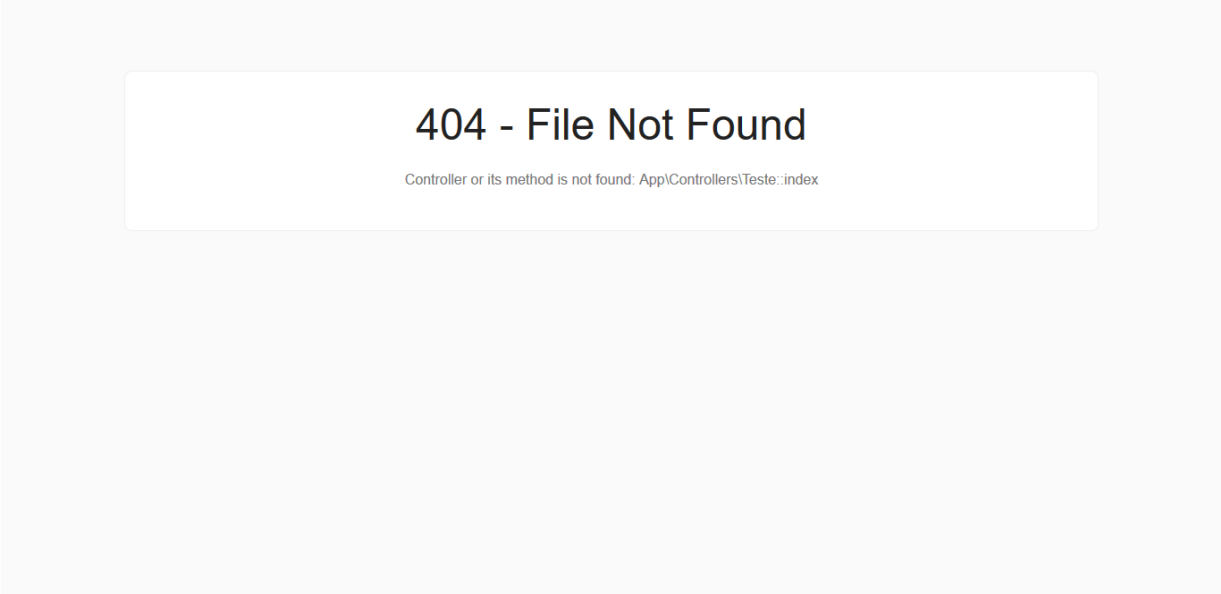


Figura 19.3: Página padrão para erro 404

ConfigException

Essa exception deve ser utilizada quando ocorrerem problemas com a classe de configuração, por exemplo, quando os valores da classe de configuração forem inválidos ou quando a classe de configuração não for do tipo correto. Essa exception fornecerá um código de status HTTP 500 e um código de saída 3.

```
throw new \CodeIgniter\Exceptions\ConfigException();
```

DatabaseException

Essa exception é lançada quando um erro de banco de dados ocorre, seja por falha de conexão ou instrução SQL incorreta ou

outros erros relacionados a banco de dados.

```
throw new \CodeIgniter\Database\Exceptions\DatabaseException();
```

Assim como em `ConfigException`, a `DatabaseException` lança o de status HTTP 500, porém o código de saída é 8.

CAPÍTULO 20
Cache de páginas

Para ajudar sua aplicação a obter o melhor desempenho, o CodeIgniter permite que você armazene em cache as páginas da sua aplicação.

Mesmo o CodeIgniter sendo bem rápido, a quantidade de informações dinâmicas exibidas em suas páginas está diretamente ligada aos recursos do servidor, memória e os ciclos de processamento utilizados, e tudo isso influencia na velocidade de carregamento da página.

Ao armazenar as páginas dinâmicas em cache, você pode obter um desempenho muito mais próximo do que pode ser visto em páginas de conteúdo estático quando estão armazenadas no cache.

Como funciona o cache

Você pode ativar o armazenamento em cache por página, assim como pode definir o tempo em que a página deve permanecer em cache antes de ser atualizada.

Quando uma página é carregada pela primeira vez, o arquivo é armazenado em cache usando o mecanismo de cache configurado no momento. Nos próximos carregamentos da página já armazenada, o arquivo em cache será recuperado e enviado ao navegador do usuário. Caso o arquivo tenha expirado, este será excluído e atualizado antes de ser enviado ao navegador.

> **ATENÇÃO**
>
> A tag `Benchmarking` não é armazenada em cache, dessa forma você pode visualizar a velocidade de carregamento da página mesmo com o cache ativado.

Ativando o cache

Para ativar o cache, basta você realizar uma chamada ao método `cachePage()` em qualquer um dos métodos do seu controlador, informando como parâmetro para ela o tempo, em segundos, pelo qual você quer manter as páginas em cache.

```
$this->cachePage($tempo_em_segundos);
```

Certifique-se de que você definiu corretamente as configurações de cache em `/app/Config/Cache.php` para que os arquivos de cache sejam gerados corretamente.

Excluindo arquivos do cache

Para interromper o armazenamento das páginas em cache basta remover a chamada ao método `cachePage()`, e quando a página expirar não será mais atualizada.

> **ATENÇÃO**
>
> Ao remover o método as páginas que ainda estão em cache permanecerão lá até expirarem. Para forçar a interrupção do uso do cache, você pode remover manualmente os arquivos de cache no diretório em que ficam armazenadas `/writable/cache` ou no diretório configurado por você em `/app/Config/Cache.php`.

Bases da aplicação

Agora que você já tem uma visão geral do CodeIgniter 4 passando pelos principais pontos no que diz respeito à estrutura e às configurações, chegou a hora de entender e aprender sobre o funcionamento do MVC dentro do framework.

Nesta terceira parte você aprenderá sobre as estruturas de controladores, views e models para começar a desenvolver suas primeiras aplicações com CodeIgniter 4, assim como algumas outras funcionalidades importantes do framework que complementam o desenvolvimento de aplicações no padrão MVC.

Capítulo 21
Controladores

Um controlador é um arquivo de classe nomeado de modo a ser associado a uma URI. Por exemplo, na URI `meusite.com.br/index.php/blog/` , o CodeIgniter buscaria pelo controlador `Blog.php` e o carregaria. Lembre-se de que o `index.php` pode ser removido da URL conforme explicado no capítulo *URLs* da Parte 2.

> O primeiro segmento da URI sempre será a identificação do controlador, exceto se você estiver trabalhando com rotas personalizadas, como veremos no capítulo *Rotas*.

21.1 Criando um controlador

Para fixar o aprendizado, nada melhor do que praticar. Então inicie um novo projeto chamado *blog_ci4* fazendo a instalação do

CodeIgniter 4 conforme explicado no capítulo *Instalação e configurações iniciais* da Parte 1.

> Você pode executar o comando `composer create-project codeigniter4/framework blog_ci --no-dev` para criar o novo projeto através da linha de comando.

Após concluir a instalação, crie um arquivo chamado `Blog.php` dentro do diretório `/app/Controller` e adicione a esse arquivo o código a seguir:

```
<?php

namespace App\Controllers;

class Blog extends BaseController
{
    public function index()
    {
        echo "Meu primeiro controlador com o CodeIgniter 4";
    }
}
```

> **ATENÇÃO** Ao salvar os arquivos de controladores, o nome do arquivo deve começar com letra maiúscula, assim como o nome da classe desse arquivo.

Esse código exibirá na tela do navegador a mensagem *Meu primeiro controlador com o CodeIgniter 4* sempre que o método `index` do controlador `Blog` for acionado.

A URL para acesso no browser pode variar de acordo com a forma como seu ambiente de desenvolvimento está estruturado, mas no geral a base da URL costuma ser a seguinte:

`http://localhost/blog_ci4` .

Lembre-se de que o CodeIgniter 4 utiliza um diretório `/public`, que é de onde devem partir os acessos, então caso você não tenha feito uma configuração para apontar direto para o diretório `/public` sua URL será `http://localhost/blog_ci4/public/index.php/blog`.

Mas se você configurou seu servidor para que ele aponte para `/public` sempre que `http://localhost/blog_ci4` for chamada no navegador, então sua URL será `http://localhost/blog_ci4/index.php/blog`.

É importante ficar atento a partir de onde o controlador está sendo estendido para que ele possa herdar todos os seus métodos do controlador pai. Nesse caso ele pode estender diretamente de `Controller` ou de `BaseController` que é um controlador estendido que já vem na instalação do framework e permite que você possa criar novas funcionalidades para os controladores, como será visto mais adiante no livro.

Definindo um controlador padrão

No CodeIgniter você pode definir um controlador padrão que será carregado quando não houver segmentos na URI, por exemplo, no caso do link principal do site.

O controlador padrão é definido em `/app/Config/Routes.php` alterando o parâmetro da variável `$routes->setDefaultController('Blog');`. E você também pode alterar o método a ser executado através da variável `$routes->setDefaultMethod('index')`.

Nesse exemplo, `Blog` é o nome da classe do controlador que você deseja utilizar e `index`, o nome do método. Caso acesse `http://localhost/blog_ci4` ou `http://localhost/blog_ci4/public` (a depender da sua configuração) o método `index()` do controlador `Blog` será executado e você verá a mensagem *Meu primeiro controlador com o CodeIgniter 4*.

Por padrão, o CodeIgniter 4 traz uma rota apontando para a raiz da aplicação e define o controlador e o método que responderão para

ela. Nesse caso, somente a troca do controlador padrão em `setDefaultController` não mudará a execução. Para que prevaleçam tanto o controlador quanto o método padrão definido, você deverá remover a definição da rota ou então modificá-la para que aponte para os desejados.

```
// Rota padrão após a instalação
$routes->get('/', 'Home::index');

// Rota modificada para ser compatível com o controlador padrão
$routes->get('/', 'Blog::index');
```

Veremos mais sobre esse assunto no capítulo *Rotas*.

Organizando os controladores em subdiretórios

Em aplicações maiores, distribuir os controladores em subdiretórios pode ser uma forma eficiente de manter a organização e até mesmo de manter uma estrutura hierárquica dos controladores. O CodeIgniter permite essa divisão e para isso basta criar os subdiretórios dentro de `/app/Controllers/` e colocar dentro deles suas respectivas classes.

Ao optar pelo uso de subdiretórios para organizar os controladores, é preciso levar em consideração que a estrutura da URI será modificada, pois nesse caso o **primeiro segmento passa a ser o nome do subdiretório**.

Tomando como base o exemplo anterior, se o controlador `Blog.php` fosse substituído pelo subdiretório `/app/Controllers/blog` e em vez de um único controlador com todas as regras relativas ao blog fossem criados múltiplos controladores, como `Posts.php`, `Categorias.php`, `Autores.php`, a URI ficaria da seguinte forma:

- **para exibir uma lista de posts:**
  `http://localhost/blog_ci4/public/index.php/blog/posts`
- **para carregar o preview de um post específico:**
  `http://localhost/blog_ci4/public/index.php/blog/posts/preview/15`

Repare que no primeiro exemplo a URI parece não ter sofrido modificações mesmo com a organização dos controladores em subdiretórios, mas no segundo exemplo os parâmetros expostos na URI começam a partir do quarto segmento e não do terceiro, como seria se os controladores estivessem na raiz do diretório `/app/Controllers`.

> **ATENÇÃO**
>
> Ao utilizar essa organização, fique atento às URIs para que os segmentos sejam estruturados corretamente e você não tenha problemas com erros por conta da recuperação de parâmetros em posições erradas. Sugiro que, se optar por trabalhar com esse tipo de organização, faça a definição das rotas em vez de utilizar o modo baseado nos segmentos da URI.

Propriedades incluídas nos controladores

Ao criar um controlador e estendê-lo de `CodeIgniter\Controller`, o novo controlador passa a ter vários recursos e propriedades disponíveis. Veja a seguir uma lista com algumas dessas propriedades e recursos:

Objeto de solicitação `$this->request`

A instância de solicitação da aplicação estará sempre disponível como uma propriedade de classe `$this->request`.

```
public function index()
{
    if ($this->request->isAJAX())
    {
        $data = [
            'success' => true,
            'id' => 123
        ];
```

```
        return $this->response->setJSON($data);
    }
}
```

Objeto de resposta `$this->response`

A principal instância de resposta da aplicação estará sempre disponível como uma propriedade de classe `$this->response`.

```
$data = 'Conteúdo do arquivo a ser disponibilizado para download.';
$name = 'nome-do-arquivo-para-download.txt';
return $this->response->download($name, $data);
```

Objetos de Log `$this->logger`

Uma instância da classe `Logger` estará disponível como uma propriedade de classe `$this->logger`.

```
$info = [
    'id'         => 85,
    'ip_address' => '192.168.1.1'
];

$this->logger->alert('Usuário #{id} logado com o IP {ip_address}', $info);
// Resultado no arquivo de log:
// ALERT - 2020-02-27 09:30:07 --> Usuário #85 logado com o IP 192.168.1.1
```

Forçar HTTPS

Um método bem conveniente para forçar o acesso via HTTPS estará disponível em todos os controladores.

```
if (! $this->request->isSecure())
{
    $this->forceHTTPS();
}
```

Carregando helpers dentro dos controladores

Você também pode definir um array de helpers como uma propriedade de classe, assim, sempre que o controlador for

carregado esses arquivos auxiliares serão automaticamente carregados na memória para que você possa usar seus métodos em qualquer lugar dentro do controlador.

```
namespace App\Controllers;
use CodeIgniter\Controller;

class Blog extends Controller
{
    protected $helpers = ['filesystem'];
}
```

21.2 Métodos

Um controlador é composto de vários métodos, que podem ser públicos ou privados. No código utilizado anteriormente foi implementado apenas o método `index()`, que sempre é carregado por padrão se o segundo segmento da URI não for informado.

Na estrutura da URI os métodos sempre serão identificados no segundo segmento, enquanto os controladores são identificados no primeiro. Mas se você estiver trabalhando com rotas personalizadas, essa sequência pode ser diferente ou nem corresponder para que a URL fique mais legível ao usuário.

Seguindo com o exemplo anterior, abra novamente o arquivo `/app/Controller/Blog.php` e crie um novo método chamado `posts()`, conforme a estrutura a seguir:

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Blog extends Controller
{

    public function index()
```

```
    {
        echo "Meu primeiro controlador com o CodeIgniter 4";
    }

    public function posts()
    {
        echo 'Esse método pode exibir uma lista de posts do blog';
    }
}
```

Salve o arquivo e abra a URL do exemplo no navegador, de acordo com a sua configuração como explicado no início do capítulo, e chame o método `posts` : `http://localhost/blog_ci4/public/index.php/blog/posts` . Na tela, você verá a mensagem *Esse método pode exibir uma lista de posts do blog*.

Recuperando parâmetros da URI no método

Muitas das vezes uma aplicação precisa de parâmetros fornecidos pelo usuário para carregar corretamente as informações, e uma das formas de enviar esses parâmetros é pela URI.

Você já sabe que o primeiro segmento da URI indica o controlador, o segundo indica o método, então os próximos podem ser utilizados para passar parâmetros.

Abra novamente o arquivo `/app/Controller/Blog.php` e atualize o método `posts()` de acordo com o código a seguir:

```
public function posts($categoria)
{
    echo "Esse método pode exibir uma lista de posts do blog na categoria
$categoria";
}
```

O parâmetro adicionado ao método corresponderá ao terceiro segmento da URI, ou seja, se sua URI for `http://localhost/blog_ci4/public/index.php/blog/posts/tecnologia` , a

mensagem exibida será *Esse método pode exibir uma lista de posts do blog na categoria tecnologia*.

Mas você pode passar mais de um parâmetro, como a quantidade de posts a serem exibidos, por exemplo.

```
public function posts($categoria, $quantidade)
{
    echo "Esse método pode exibir uma lista de $quantidade posts do blog na categoria $categoria";
}
```

A URI para essa situação seria `http://localhost/blog_ci4/public/index.php/blog/posts/tecnologia/5` , onde a quantidade estaria indicada no quarto segmento e o resultado seria *Esse método pode exibir uma lista de 5 posts do blog na categoria tecnologia*.

> **ATENÇÃO**
>
> Caso você esteja utilizando o roteamento de URIs, os segmentos passados para o método serão os redirecionados.

Métodos privados e protegidos

Em alguns casos, é conveniente que alguns métodos sejam acessados apenas por dentro da aplicação, isto é, que eles não posam ser acessados via URI no navegador. Se você tentar acessar a URI `http://localhost/blog_ci4/public/index.php/blog/validar_posts/` e o método estiver construído como no exemplo a seguir, você não conseguirá nenhum resultado exceto um erro 404.

```
protected function existe_post()
{
    // código da verificação
}

private function validar_post(){
```

```
    // código da validação
}
```

Os métodos privados ou protegidos possuem algumas diferenças:

- **métodos protegidos:** podem ser acessados na classe que os declarou e nas classes herdeiras da classe declarante.
- **métodos privados:** só podem ser acessados na classe que os declarou.

Essas diferenças garantem maior segurança para a sua aplicação e de uma certa forma ajudam a manter o código organizado.

Acesse o repositório do livro no GitHub e veja todos os exemplos de código apresentados aqui.

https://github.com/jlamim/livro-codeigniter4

Capítulo 22
Filtros de controladores

Os filtros de controladores permitem executar ações antes e depois da execução dos mesmos. Você pode escolher quais URIs em sua aplicação terão filtros aplicados e assim fazer com que as solicitações sejam modificadas quando os filtros forem executados *antes* da execução dos controladores. Ou então, que as respostas sejam modificadas quando os filtros forem executados *após* os controladores.

Veja alguns exemplos de tarefas que podem ser executadas com filtros:

- Aplicar proteção CSRF nas solicitações recebidas
- Restringir acesso a áreas do site com base nas permissões de usuário
- Exibir "Página em manutenção"
- Executar negociação automática de conteúdo

22.1 Criando um filtro

Os filtros são classes que implementam `CodeIgniter\Filters\FilterInterface` e possuem dois métodos, `before()` e `after()`, que recebem os códigos a serem executados antes e depois do controlador, respectivamente. Caso um dos métodos não seja necessário, ele pode ficar em branco, sem código. Os filtros criados devem ser armazenados em `/app/Filters`.

Veja a seguir um exemplo de uma classe de filtro de controlador:

```
<?php namespace App\Filters;
```

```php
use CodeIgniter\HTTP\RequestInterface;
use CodeIgniter\HTTP\ResponseInterface;
use CodeIgniter\Filters\FilterInterface;

class Filtro implements FilterInterface
{
    public function before(RequestInterface $request)
    {
        // Código a ser executado
    }

    //--------------------------------------------------------------------

    public function after(RequestInterface $request, ResponseInterface
$response)
    {
        // Código a ser executado
    }
}
```

Filtro `before()`

Em qualquer filtro, você pode retornar o objeto `$request`, substituindo a solicitação atual. Dessa forma, você tem permissão para realizar alterações que ainda estarão presentes quando o controlador for executado.

Como você está realizando uma operação antes de o controlador ser executado, é possível impedir que as ações no controlar aconteçam, e isso pode ser feito devolvendo qualquer coisa que não seja o objeto da solicitação. É comum utilizar esse recurso para realizar redirecionamentos.

Pense em uma situação onde alguns conteúdos de um blog são de acesso restrito. Você pode utilizar o filtro `before()` para verificar se o usuário está logado e, caso não esteja, o redirecionar para a página de login.

```php
public function before(RequestInterface $request)
{
    $auth = service('auth');

    if (! $auth->isLoggedIn())
    {
        return redirect('login');
    }
}
```

Filtro `after()`

Os filtros `after()` são quase idênticos aos filtros `before()`, a diferença está no que é possível retornar, já que no filtro `after()` você retorna apenas o objeto `$response` e não pode parar a execução do script. Isso permite que a saída final seja modificada ou que você faça alguma coisa com essa saída.

Uma situação em que o filtro `after()` poderia ser utilizado seria para garantir que determinados cabeçalhos de segurança fossem definidos de maneira correta, ou para armazenar em cache a saída final, ou até mesmo filtrar conjuntos de palavras no conteúdo a ser exibido na tela ou mesmo armazenar uma informação no log.

```php
public function after(RequestInterface $request, ResponseInterface
$response)
{
    // Aqui é gravado um log com as informações da requisição recebida
    log_message('info', json_encode($request));
}
```

22.2 Configurando filtros

Após criar os filtros você precisa configurar o momento em que eles serão executados. Essa configuração é feita em `/app/Config/Filter.php`. Nesse arquivo, você encontrará quatro

propriedades que permitem a configuração do momento exato em que os filtros serão executados.

```
<?php namespace Config;

use CodeIgniter\Config\BaseConfig;

class Filters extends BaseConfig
{

 public $aliases = [];

 public $globals = [
  'before' => [],
  'after'  => [],
 ];

 public $methods = [];

 public $filters = [];
}
```

$aliases

O array `aliases` é utilizado para associar nomes simples ou nomes de classes devidamente qualificados, que são os filtros a serem executados:

```
public $aliases = [
    'csrf'     => \CodeIgniter\Filters\CSRF::class,
    'toolbar'  => \CodeIgniter\Filters\DebugToolbar::class,
    'honeypot' => \CodeIgniter\Filters\Honeypot::class,
];
```

Os aliases são obrigatórios e caso você tente utilizar o nome completo da classe posteriormente, a aplicação emitirá um erro.

Defini-los dessa maneira simplifica a troca da classe usada. É ótimo para quando você decide mudar para um sistema de autenticação

diferente, por exemplo, pois basta alterar apenas a classe do filtro e está pronto.

É possível combinar vários filtros em um alias, simplificando a aplicação de conjuntos complexos de filtros:

```
public $aliases = [
    'apiPrep' => [
        \App\Filters\Negotiate::class,
        \App\Filters\ApiAuth::class
    ]
];
```

`$globals`

Em `globals`, você definirá quais filtros devem ser aplicados às solicitações feitas pela estrutura. Tome cuidado com a quantidade de filtros utilizados aqui, pois pode afetar o desempenho da aplicação se muitos forem executados em cada solicitação. Os filtros podem ser especificados adicionando seu alias ao array `before` ou `after`.

```
public $globals = [
    'before' => [
        'honeypot'
        'csrf',
    ],
    'after'  => [
        'toolbar',
    ],
];
```

Pode acontecer de em algum momento você precisar aplicar filtros somente em uma parte das solicitações. Um bom exemplo é em relação à aplicação de proteção CSRF, onde pode ser necessário excluir alguns URIs do filtro de modo a permitir que solicitações de sites de terceiros acessem apenas URIs específicos, mantendo o restante protegido.

Para fazer isso, adicione um array com `except` e um URI correspondente ao lado do `alias`. No exemplo a seguir, toda URL que começar com `api/` não passará pelo filtro de proteção CSRF.

```
public $globals = [
    'before' => [
        'csrf' => ['except' => 'api/*']
    ],
    'after'  => []
];
```

Sempre que você precisar especificar uma URI nas configurações de filtro, pode utilizar expressões regulares ou o asterisco (`*`), que funciona como um coringa, correspondendo a todos os caracteres a partir do ponto em que está.

Você pode utilizar várias URIs quando precisar especificá-las, basta colocá-las em um array:

```
public $globals = [
    'before' => [
            'csrf' => ['except' => ['foo/*', 'bar/*']]
    ],
    'after'  => []
];
```

`$methods`

Os filtros também podem ser aplicados a solicitações baseadas em métodos HTTP, como `POST`, `GET`, `PUT` e outros. Basta especificar no array `$methods` o nome do método com letra minúscula. Diferente das propriedades `$globals` e `$filters`, aqui só há execução antes dos filtros:

```
public $methods = [
    'post' => ['foo', 'bar'],
    'get'  => ['baz']
]
```

Além dos métodos HTTP padrão, são suportadas outras duas situações: `cli` e `ajax`, ou seja, é possível aplicar filtros para requisições feitas via linha de comando e AJAX, respectivamente.

`$filters`

Essa propriedade é um array de aliases de filtros, sendo que para cada alias você pode especificar arrays `before` e `after` contendo uma lista de padrões de URI aos quais o filtro deve ser aplicado.

```
public filters = [
    'AcessoBlog' => ['before' => ['blog/index'], 'after' =>
['blog/posts']]
];
```

Capítulo 23
Rotas

Toda aplicação possui uma relação entre uma URL e uma combinação de controlador, método e parâmetros, e essa combinação compõe o que chamamos de URI, que possui a seguinte representação:

```
meusite.com.br/controlador/metodo/parametros
```

Existirão situações em que será necessário modificar a estrutura da URI para que um controlador ou método diferente seja chamado ou mesmo para que ela fique mais legível para o usuário.

Imagine que o link para a leitura dos posts de um blog deva ser composto apenas pelo domínio do site, a indicação de que é conteúdo do blog e o título do post, nada mais.

```
meusite.com.br/blog/titulo-do-meu-post-no-blog
```

Nesse caso temos uma URI diferente do padrão `meusite.com.br/controlador/metodo/parametros`, mas isso não é problema pois através das rotas podemos apontar esse padrão de URI para o seu respectivo controlador e método.

23.1 Definindo as regras de roteamento

As regras de roteamento são definidas em `/app/Config/Routes.php`. Ele é uma instância da classe `RouteCollection`, que permite especificar seus próprios critérios de roteamento.

A seguir você pode ver a configuração padrão de roteamento após a criação de um novo projeto usando o CodeIgniter 4.

```
<?php namespace Config;

$routes = Services::routes(true);

if (file_exists(SYSTEMPATH . 'Config/Routes.php'))
{
    require SYSTEMPATH . 'Config/Routes.php';
}

$routes->setDefaultNamespace('App\Controllers');
$routes->setDefaultController('Home');
$routes->setDefaultMethod('index');
$routes->setTranslateURIDashes(false);
$routes->set404Override();
$routes->setAutoRoute(true);

$routes->get('/', 'Home::index');

if (file_exists(APPPATH . 'Config/' . ENVIRONMENT . '/Routes.php'))
{
    require APPPATH . 'Config/' . ENVIRONMENT . '/Routes.php';
}
```

O objeto `RouteCollection` permite modificar a maneira como o roteador está funcionando, atuando como um suporte para as definições de configuração. Veja a seguir alguns métodos que podem ser chamados no objeto para modificar as operações padrão:

- `$routes->setDefaultNamespace()` : modifica o namespace adicionado a um controlador se não existir um.
- `$routes->setDefaultController()` : altera o nome da classe utilizada como controlador padrão da aplicação, que é chamado quando nenhuma classe é especificada no primeiro segmento da URI.
- `$routes->setDefaultMethod()` : atribui o método dentro do controlador que é executado quando o roteador não consegue

determinar o método a ser executado.

- `$routes->setTranslateURIDashes()` : essa opção permite a substituição automática de traços (`-`) por sublinhados nos segmentos do URI do controlador e método. Uma vez que traços não são caracteres válidos para nomes de controladores e métodos, é possível substituí-los por underline (_), evitando a necessidade de criação de rotas alternativas.
- `$routes->set404Override()` : determina a rota que deve ser executada quando um erro 404 acontecer na aplicação. Caso não seja definida uma rota, será apresentada a página padrão do CodeIgniter.
- `$routes->setAutoRoute()` : determina se o roteador tentará a correspondência de URIs aos controladores quando nenhuma rota específica tiver sido definida. Caso o método receba como parâmetro `false` , apenas rotas já definidas estarão disponíveis na aplicação.

Tomando como base o código `$routes->get('/', 'Home::index');` temos a seguinte estrutura na criação de uma rota:

- `$routes` : objeto contendo os métodos e necessários para trabalhar com a rotas, instanciado em `$routes = Services::routes(true);` .
- `get` : método HTTP pelo qual a rota responderá.
- `'\'` : parâmetro que corresponde à estrutura da URI que será utilizada pelo usuário para acessar uma página.
- `Home::index` : controlador e método, respectivamente, separados por `::` que responderão pela URI acessada.

> `Home::index` é o equivalente a `$home->index()` .

23.2 Placeholoders

Os placeholders, ou espaços reservados, são cadeias de caracteres que representam um padrão de *expressão regular*. Ao processar as rotas, o CodeIgniter substitui esses placeholders pelo valor da expressão regular.

Para visualizar melhor o uso de placeholders, vamos utilizar como base a URL `meusite.com.br/blog/titulo-do-meu-post-no-blog`. Uma estrutura de rota a ser adotada para responder a ela seria:

```
$routes->get('/blog/(:any)', 'Blog::ler/$1');
```

`(:any)` é um placeholder que suporta todos os caracteres válidos para URLs. Para recuperar o seu valor e passá-lo como parâmetro para o método é utilizado `$1` logo após a definição do controlador e do método que responderão à requisição.

Em uma rota, o primeiro parâmetro contém o URI a ser correspondido e o segundo contém o destino para o qual ele deve ser roteado novamente.

No exemplo acima, se a palavra literal *blog* for encontrada no primeiro segmento da URL e letras ou números (ou ambos) forem encontrados no segundo segmento, a classe `Blog` e o método `ler` serão chamados e o método `ler` receberá como parâmetro o valor do segmento que foi definido com o placeholder.

Veja a seguir uma lista com os placeholders disponíveis:

- `(:any)` : corresponderá a todos os caracteres desse ponto até o final do URI, podendo incluir vários segmentos de URI.
- `(:segment)` : corresponderá a qualquer caractere, exceto a barra (`/`), que restringe o resultado a um único segmento.
- `(:num)` : corresponderá a qualquer número inteiro.
- `(:alpha)` : corresponderá a qualquer sequência de caracteres alfabéticos.
- `(:alphanum)` : corresponderá a qualquer sequência de caracteres alfabéticos ou números inteiros ou combinação dos dois.

- `(:hash)` : é o mesmo que `:segment` , mas pode ser utilizado para ver de forma mais fácil quais rotas usam IDs de hash.

> **ATENÇÃO**
>
> `{locale}` não pode ser utilizado como um placeholder ou qualquer outra parte da rota, pois é uma palavra reservada do framework.

Exemplos de rotas com placeholders

Veja a seguir alguns exemplos de roteamento utilizando placeholders.

```
$routes->add('artigos', 'Blog');
$routes->add('blog/(:any)', 'Blog::ler');
$routes->add('blog/categoria/(:any)/(:num)', 'Blog::posts/$1/$2');
```

Embora o método `add()` seja muito conveniente, a recomendação é utilizar sempre as rotas baseadas em verbos HTTP, como será apresentado a seguir, pois elas são mais seguras.

Utilizar rotas baseadas em verbos HTTP também produz um aumento no desempenho da aplicação, uma vez que apenas as rotas correspondentes ao método de solicitação atual são armazenadas, o que resulta em uma quantidade menor de rotas a serem analisadas para encontrar a correspondência.

Exemplos de rotas utilizando verbos HTTP

```
$routes->get('artigos', 'AdminBlog::listPosts');
$routes->post('artigo', 'AdminBlog::addPost');
$routes->put('artigo/(:num)', 'AdminBlog::updatePost/$1');
$routes->delete('artigo/(:num)', 'AdminBlog::removePost/$1');
```

> Esse tipo de rota é de grande utilidade na criação de APIs.

Placeholders customizados

É possível criar seus próprios placeholders para utilizar na sua aplicação, personalizando a experiência e a legibilidade.

Placeholders customizados são adicionados com o método `addPlaceholder()`, cujo primeiro parâmetro é a sequência a ser usada como placeholder e o segundo é o padrão de expressão regular pelo qual deve ser substituído. O método `addPlaceholder()` deve ser chamado antes de a rota ser adicionada.

```
$routes->addPlaceholder('uuid', '[0-9]{4}-[a-f]{4}');
$routes->add('blog/ler/(:uuid)', 'Blog::getPostByUUID/$1');
```

Placeholder com expressões regulares

Se você preferir utilizar expressões regulares, ou se não for o caso de preferência mas necessidade, o mapeamento de rotas do CodeIgniter permite utilizá-las, desde que seja uma expressão regular válida, assim como as referências mostradas anteriormente.

Veja no exemplo a seguir uma URI para listagem de posts de uma categoria específica (segmento 2) e uma página específica (segmento 3):

```
$routes->add('posts/([a-z]+)/(\d+)', 'Blog::posts/$1/$2');
```

Através das expressões regulares você pode capturar um segmento de URI que contém uma barra (`/`), o que normalmente representaria o delimitador entre vários segmentos.

Por exemplo, se um usuário acessa uma área da aplicação protegida por senha e você quer redirecioná-lo para a mesma página após o login, a seguinte rota seria de muita utilidade:

```
$routes->add('login/(.+)', 'Auth::login/$1');
```

Se você não tem muita experiência com expressões regulares, o site http://regular-expressions.info é um ótimo ponto de partida.

Cuidados a se tomar ao construir rotas utilizando placeholders

Com ou sem placeholder as rotas sempre manterão sua estrutura baseada em segmentos, e dependendo da forma como você construir as rotas e a ordem em que as declarar no arquivo `/app/Config/Routes.php` pode acontecer de as rotas não serem executadas corretamente, mesmo que a definição esteja correta.

Veja o exemplo de declaração de rotas a seguir:

```
$routes->addPlaceholder('uuid', '[0-9]{4}-[a-f]{4}');

$routes->get('/', 'Blog::index');
$routes->get('blog/uuid/(:uuid)', 'Blog::getPostByUUID/$1');
$routes->get('blog/(:any)', 'Blog::ler/$1');
$routes->get('blog/categoria/(:any)/(:num)', 'Blog::posts/$1/$2');

$routes->add('autores/perfil', 'Blog::autor', ['as' => 'perfil']);
// Redirecionamento para o nome da rota
$routes->addRedirect('autor/sobre', 'perfil');
// Redirecionamento para o URI
$routes->addRedirect('autor/sobre', 'autores/perfil');

$routes->environment('development', function ($routes) {
    $routes->add('logs', 'Log::index');
});
```

Temos uma estrutura de rotas definidas corretamente, apontando para seus respectivos controladores e métodos. Uma das rotas definidas usando placeholder é `$routes->get('blog/(:any)', 'Blog::ler/$1');` , que é a rota responsável por carregar um post para a leitura, como vimos no início do capítulo.

Se analisar bem a estrutura da rota, ela é parecida com a estrutura automática `controlador/método` , porém o nome do método é substituído por um placeholder e isso faz com que, ao tentar executar uma rota automática que não foi declarada, o CodeIgniter considere a rota declarada usando placeholder e não o método existente no controlador.

Por exemplo, se eu tenho um método no controlador `Blog` chamado `https` que força um redirecionamento para a URL acessada sob o protocolo HTTPS e chamo a URI `meusite.com.br/blog/https`, a aplicação vai responder a partir da rota `$routes->get('blog/(:any)', 'Blog::ler/$1');` pois o segundo segmento corresponde ao placeholder. Sendo assim, o método chamado no controlador será o `ler` e não o `https`.

Esse problema é muito comum, e às vezes faz com que o desenvolvedor gaste um bom tempo procurando onde está o problema. Minha sugestão para você evitar isso é declarar todas as rotas utilizadas pela aplicação, assim você terá um melhor controle e uma visualização mais ampla sobre os controladores e métodos que são chamados para cada rota.

A seguir você verá diversas formas que ajudarão você a organizar as rotas e gerenciá-las de forma simples e rápida.

23.3 Mapeamento e redirecionamento de rotas

Mapeando múltiplas rotas

Embora o método `add()` seja simples de usar, em geral é mais fácil trabalhar com várias rotas ao mesmo tempo usando o método `map()`.

Em vez de chamar `add()` para cada rota a ser adicionada, você pode definir um array de rotas e passá-lo como o primeiro parâmetro do método `map()`:

```
$routes = [];
$routes['blog/post/(:any)']  = 'Blog::post';
$routes['blog/post/(:num))'] = 'Blog::postByID/$1';

$collection->map($routes);
```

Redirecionando rotas

Uma rotina muito comum em sites, blog, lojas virtuais e até mesmo sistemas é ter páginas mudando de endereço. O redirecionamento de rotas do CodeIgniter é capaz de interpretar esse caminho e não deixar que sua aplicação retorne erro 404 quando uma rota não for encontrada porque mudou.

A implementação do redirecionamento é feita com o método `addRedirect()`, cujo primeiro parâmetro é o padrão URI para a rota antiga, o segundo é o novo URI ou o nome da rota para o qual deve acontecer o redirecionamento e o terceiro parâmetro é o código de status HTTP que deve ser enviado junto ao redirecionamento. O valor padrão é 302, que é um redirecionamento temporário e é recomendado na maioria dos casos, então se não for informado o parâmetro esse será o valor utilizado.

```
$routes->add('autores/perfil', 'Blog::autor', ['as' => 'perfil']);

// Redirecionamento para o nome da rota
$routes->addRedirect('autor/sobre', 'perfil');
// Redirecionamento para o URI
$routes->addRedirect('autor/sobre', 'autores/perfil');
```

Caso haja correspondência entre a rota acessada pelo usuário e uma rota de redirecionamento, o usuário será redirecionado de imediato para a nova página antes mesmo do carregamento do controlador associado à rota antiga.

Agrupamento de rotas

O agrupamento de rotas é uma forma de organizar as rotas de nome em comum através do método `group()`.

```
$routes->group('blog', function($routes)
{
    //URI: blog/autores
    $routes->add('autores', 'Blog::autores');
    //URI: blog/posts
```

```
    $routes->add('posts', 'Blog::posts');
    //URI: blog/configuracoes
    $routes->add('configuracoes', 'Blog::configuracoes');
});
```

O nome do grupo, que é o primeiro parâmetro a ser passado no método `group()`, se torna um segmento que aparece antes das rotas definidas dentro do grupo, permitindo a redução da digitação necessária para criar um conjunto extenso de rotas que compartilham a sequência de abertura, como no exemplo acima, onde foram agrupadas rotas de um painel administrativo de um blog.

Também é possível aninhar grupos dentro de grupos para uma organização melhor caso necessário.

```
$routes->group('blog', function($routes)
{
    $routes->group('posts', function($routes)
    {
        $routes->add('listar', 'Blog::posts');
        $routes->add('editar', 'Blog::posts_editar');
        $routes->add('novo', 'Blog::posts_novo');
    });
});
```

Se por algum motivo você precisar atribuir opções para um grupo, como um namespace, faça isso antes do retorno da chamada.

```
$routes->group('api', ['namespace' => 'App\API\v1'], function($routes)
{
    $routes->resource('blog');
});
```

Nesse exemplo há uma manipulação da rota de recurso para o controlador `App\API\v1\Blog` com o URI `/api/blog`.

Você também pode usar um filtro específico para um grupo de rotas. Esse filtro sempre será executado antes ou depois do controlador e é muito útil na autenticação ou registro de APIs.

```
$routes->group('api', ['filter' => 'apiCheck'], function($routes)
{
    $routes->resource('blog');
});
```

O valor do filtro deve ser correspondente a um dos aliases definidos em `/app/Config/Filters.php`.

Closures

Você também pode fazer uso de função anônima, ou Closure, como destino para uma rota. A função definida será executada sempre que houver solicitação pelo URI.

Closure é muito útil para executar de forma rápida pequenas tarefas, ou mesmo mostrar apenas uma `view` simples como um feed de conteúdo, por exemplo:

```
$routes->add('feed', function()
{
    $rss = new RSSFeeder();
    return $rss->feed('general');
});
```

23.4 Restrição de rotas

Por ambientes

Quando se está desenvolvendo uma aplicação é comum ter outros ambientes além do de desenvolvimento, como teste e produção, e o CodeIgniter permite criar rotas restritas a um determinado ambiente através do método `environment()`.

Essa possibilidade de restringir rotas por ambiente é útil para manter recursos que só podem ser utilizados em ambientes específicos.

Como exemplo, temos os *debugs*, que só fazem sentido em ambiente de desenvolvimento e às vezes em ambiente de teste.

O primeiro parâmetro a ser passado para o método `environment()` é o nome do ambiente em que as rotas serão limitadas a funcionar. O segundo é a resposta contendo as rotas que ficarão disponíveis apenas no ambiente informado no primeiro parâmetro.

```
$routes->environment('development', function($routes)
{
    $routes->add('logs', 'Logs::index');
});
```

Por domínio ou subdomínio

Você pode restringir grupos de rotas para que funcionem apenas em determinados domínios ou subdomínios da aplicação. Para isso, basta informar como último parâmetro uma matriz com o domínio especificado em `hostname` ou, se for uma restrição por subdomínio, especificada em `subdomain` .

```
$collection->get('from', 'to', ['hostname' => 'seudominio.com']);
```

Para subdomínios o CodeIgniter vai considerar o domínio em que a aplicação está sendo executada para então aplicar a restrição do subdomínio. Também há a possibilidade de utilizar o caractere curinga (`*`) para liberar a rota para qualquer subdomínio.

```
$routes->add('from', 'to', ['subdomain' => 'seusubdominio']);
$routes->add('from', 'to', ['subdomain' => '*']);
```

> **ATENÇÃO**
>
> Ao implementar esse recurso de restrição por subdomínio, faça testes no ambiente de produção pois pode acontecer de alguns domínios produzirem o *falso positivo* dependendo de como seus nomes foram registrados.

Caso o usuário tente acessar uma rota e ela esteja restrita para ele, será retornado um erro 404 com sua respectiva página.

23.5 Roteamento reverso e rotas nomeadas

Roteamento reverso

O roteamento reverso permite a definição do controlador e do método, bem como seus parâmetros aos quais um link deve ir e fazer com que o roteador procure a rota atual para ele.

Esse recurso permite que as definições de rota sejam modificadas sem a necessidade de você atualizar o código da aplicação. Seu uso é muito comum nas views para a criação de links.

Se você tem uma rota para uma galeria de fotos à qual deseja criar um link, pode utilizar a função auxiliar `route_to()` para obter a rota atual que deve ser utilizada.

Na função `route_to()` o primeiro parâmetro é o controlador e o método devidamente qualificados, separados por duplo dois-pontos (`::`), da mesma forma como é utilizado para escrever a rota inicial. Os demais parâmetros a serem enviados à rota são passados na sequência.

```
// Definição da rota
$routes->add('blog/post/(:id)/galeria/(:any)',
'App\Controllers\Galerias::visualizarGaleriaPost/$1/$2');

// URL relativa para o post 15 e a galeria 12
// /posts/15/galeria/12
<a href="<?= route_to('App\Controllers\Galerias::visualizarGaleriaPost',
15, 12) ?>">Visualizar Galeria de Fotos</a>
```

Rotas nomeadas

É possível nomear rotas para tornar a aplicação menos frágil quanto a links quebrados. Quando você aplica um nome a uma rota, esse pode ser chamado a qualquer momento dentro da aplicação, desde que `route_to()` esteja sendo utilizada na criação dos links. Assim, se você precisar alterar a definição da rota, fará isso apenas em `/app/Config/Routes.php` e o método `route_to()` se encarregará de manter todos os links atualizados em relação a essa mudança.

A principal alteração em relação ao roteamento reverso é que o método `add()` receberá um terceiro parâmetro, que é um array com a definição do nome da rota. Na view, ao chamar o método `route_to()`, você informará apenas o nome da rota definido em `as` e os parâmetros.

```
// Definição da rota
$routes->add('blog/post/(:id)/galeria/(:any)',
'App\Controllers\Galerias::visualizarGaleriaPost/$1/$2', ['as' =>
'post_galeria']);

// URL relativa para o usuário ID 15 e a galeria ID 12
// /posts/15/galeria/12
<a href="<?= route_to('post_galeria', 15, 12) ?>">Visualizar Galeria de
Fotos</a>
```

Além de tornar a codificação mais legível, o uso de rotas nomeadas torna a inserção de código na view mais fácil, principalmente se quem estiver trabalhando na construção das views tiver pouca experiência com CodeIgniter.

Parâmetros opcionais

Todos os métodos apresentados neste capítulo podem fazer uso de um conjunto variado de parâmetros opcionais capazes de modificar as rotas ou restringi-las ainda mais, como por exemplo `offset`, `as`, `redirect`, `hostname` e `subdomain`. Sempre que for necessário fazer uso desses parâmetros de configuração, eles devem ser passados como último parâmetro do método, na forma de array, como pode ser visto a seguir:

```php
$collection->get('from', 'to', ['hostname' => 'seudominio.com']);
$routes->add('from', 'to', ['subdomain' => 'seusubdominio']);
$routes->add('autores/perfil', 'Blog::autor', ['as' => 'perfil']);
```

Capítulo 24
Views

As views no CodeIgniter são páginas web ou fragmentos de páginas web que podem ser incorporadas a outras páginas da aplicação para compor a estrutura visual que será acessada pelo usuário.

Views nunca são chamadas diretamente, são sempre carregadas por um controlador, que dentro do MVC atua como responsável por controlar quais views serão chamadas e que informações elas receberão para serem exibidas aos usuários.

24.1 Criando as views

Para criar uma view você deve criar o seu arquivo com a estrutura HTML dentro do diretório `/app/Views`, nomeando-o preferencialmente com o nome do método ao qual essa view corresponde. Dessa forma, a estrutura de código fica mais legível e organizada.

No capítulo sobre **Controladores** você criou um projeto chamado `blog_ci4`, com controlador e alguns métodos. Agora você vai começar a dar mais corpo a esse projeto criando algumas views.

Crie um arquivo dentro do diretório `/app/Views/Blog` e chame-o de `Index.php`. Em seguida adicione a ele o código a seguir, que é uma estrutura HTML simples.

```
<html>
<head>
    <title>Livro CodeIgniter 4</title>
</head>
<body>
    <h1>Minha primeira view com o CodeIgniter 4</h1>
```

```
</body>
</html>
```

24.2 Exibindo as views

Para carregar e exibir uma view você deve utilizar a função `view()` passando como parâmetro o nome da sua view, sem a necessidade de especificar a extensão do arquivo.

```
echo view('Blog/Index');
```

Agora abra o arquivo `/app/Controller/Blog.php` e, no método `index()`, substitua a linha que exibe a mensagem *Meu primeiro controlador com o CodeIgniter 4* pela chamada à view que você acabou de criar.

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Blog extends Controller
{
    public function index()
    {
        echo view('Blog/Index');
    }
}
```

No CodeIgniter, você pode carregar múltiplas views em um mesmo método e a estrutura da página a ser visualizada pelo usuário será construída conforme a sequência em que as views forem chamadas no método.

Para exemplificar essa estrutura, separe a view `Blog/Index.php` em 3 partes:

cabeçalho: `/app/Views/Cabecalho.php`

```
<html>
<head>
    <title>CodeIgniter 4</title>
</head>
<body>
```

corpo da página: `/app/Blog/Index.php`

```
<h1>Minha primeira view com o CodeIgniter 4</h1>
```

rodapé: `/app/Views/Rodape.php`

```
</body>
</html>
```

Com as views devidamente separadas, atualize o método `index()` no controlador `Blog` para que ele possa chamar corretamente a estrutura de views que compõe a página.

```
public function index()
{
    echo view('Cabecalho');
    echo view('Blog/Index');
    echo view('Rodape');
}
```

Dessa forma você tem as estruturas independentes e não precisa ficar repetindo em todas as views o cabeçalho e rodapé padrão da aplicação.

24.3 Adicionando as views ao cache

As views podem ser armazenadas em cache para otimizar o processo de carregamento das páginas através do uso de parâmetros de configuração, que são o terceiro parâmetro do método `view()` . Para armazenar uma view ao cache por 60 segundos basta chamar a view da seguinte forma:

```
echo view('nome_da_view', $dados, ['cache' => 60);
```

O CodeIgniter vai armazenar a view no cache com o mesmo nome do arquivo, mas você pode customizar o nome passando o parâmetro `cache_name` no array que compõe o terceiro parâmetro do método `view()`.

```
echo view('nome_da_view', $dados, ['cache' => 60, 'cache_name' =>
'nome_da_view_no_cache']);
```

Adicione as views `/app/Views/Cabecalho.php` e `/app/Views/Rodape.php` ao cache alterando a chamada a elas em `/app/Controllers/Blog`.

```
public function index()
{
    echo view('Cabecalho', [], ['cache' => 60, 'cache_name' =>
'cabecalho_cache']);
    echo view('Blog/Index');
    echo view('Rodape', [], ['cache' => 60, 'cache_name' =>
'rodape_cache']);
}
```

24.4 Adicionando dados dinâmicos às views

Em sua grande maioria, as views exibem para os usuários os dados que foram solicitados, devidamente formatados e organizados.

Mas esses dados são dinâmicos e precisam ser passados às views também de forma dinâmica, e isso é feito utilizando o segundo parâmetro do método `view()`, que é o responsável por agrupar em um array todos os dados a serem exibidos.

Atualize o método `index()` no controlador `Blog` conforme o código a seguir para criar o array com as informações dinâmicas que serão passadas para as views, e apontar o segundo parâmetro no método `view()`.

```
public function index()
{
    $dados = [
        "titulo_da_pagina" => "Livro CodeIgniter 4",
        "mensagem"         => "Minha primeira view com o CodeIgniter 4"
    ];
    echo view('Cabecalho', $dados, ['cache' => 60, 'cache_name' =>
'cabecalho_cache']);
    echo view('Blog/Index', $dados);
    echo view('Rodape', [], ['cache' => 60, 'cache_name' =>
'rodape_cache']);
}
```

Agora atualize a estrutura das views `Cabecalho.php` e `Blog/Index.php` para que as informações dinâmicas sejam inseridas junto à estrutura HTML.

Cabecalho. php

```
<html>
<head>
    <title><?= $titulo_da_pagina ?></title>
</head>
<body>
```

/app/Blog/Index. php

```
<h1><?= $mensagem ?></h1>
```

Repare que os textos foram substituídos por variáveis que correspondem às chaves do array `$dados` que foi criado no controlador `Blog` e transmitido às views através do segundo parâmetro do método `view()` .

Assim como foram enviadas para as views informações textuais, poderiam ter sido enviados outros tipos de dados e até mesmo conjuntos de dados para serem exibidos na view através de

loopings, por exemplo, uma lista de informações e registros para o usuário.

Desde que as informações dinâmicas a serem transmitidas para as views estejam organizadas em um array, você pode passar o que for necessário, com os devidos tratamentos, e na view apenas organizar a exibição.

24.5 Montando views fora do controlador

As *View Cells* são uma forma muito útil de organizar blocos de código para serem gerados fora dos controladores. Através de uma classe e um método específicos que retornam uma string HTML válida a ser injetada dentro da view, você pode tornar ainda melhor e mais eficiente a modularização das suas aplicações em CodeIgniter.

Os métodos que retornam strings HTML para exibição em células precisam estar devidamente estruturados de modo que o *autoload* da aplicação possa encontrá-los e carregá-los.

> **ATENÇÃO**
>
> Existe uma única restrição para a classe que contém métodos que retornam conteúdo para *View Cells*: **ela não pode ter nenhum parâmetro de construtor**.

Um exemplo bastante interessante do uso de *View Cells* é a exibição de blocos com o mesmo layout. Esses blocos podem aparecer em diversas páginas e ter seu conteúdo definido com base na página em questão.

Para utilizar as *View Cells* basta chamar o método `view_cell()` de modo semelhante ao que foi feito com o método `view()`, porém, no

`view_cell()` o primeiro parâmetro corresponde ao método que retorna a string HTML válida e o segundo parâmetro, ao array contendo os parâmetros para o método a ser executado.

```
//chamada sem parâmetros
view_cell('\App\Libraries\Blog::postsRecentes');

//chamada com parâmetros
view_cell('\App\Libraries\Blog::postsRecentes', ['categoria' =>
'tecnologia', 'max_posts' => 3]);
```

Mais adiante veremos como criar bibliotecas e helpers no CodeIgniter 4, mas para você compreender esse exemplo, o método `postsRecentes()` ficaria com uma estrutura semelhante à exibida a seguir.

```
public function postsRecentes(array $parametros=[])
{
    $posts = $this->blogModel->where('categoria',
$parametros['categoria'])
                             ->orderBy('publicado_em', 'desc')
                             ->limit($params['max_posts'])
                             ->get();

    return view('Blog/postsRecentes', ['posts' => $posts]);
}
```

Nesse método os dados são obtidos diretamente no model com base nos parâmetros informados na chamada ao método. O retorno do método `postsRecentes()` é uma view que foi chamada usando o método `view()` e é capaz de renderizar os dados passados a ela como parâmetro.

Há uma outra maneira até mais legível de passar os parâmetros para o método `postsRecentes()`. Nela, em vez de passar um único parâmetro no formato de array são passados parâmetros individuais, como pode ser visto no exemplo a seguir:

```
view_cell('\App\Libraries\Blog::postsRecentes', 'categoria = tecnologia,
max_posts = 3');
```

```
public function postsRecentes(string $categoria, int $max_posts)
{
    $posts = $this->blogModel->where('categoria',
$parametros['categoria'])
                             ->orderBy('publicado_em', 'desc')
                             ->limit($params['max_posts'])
                             ->get();

    return view('Blog/postsRecentes', ['posts' => $posts]);
}
```

Colocando View Cells no cache

Também é possível adicionar as *View Cells* no cache, e para isso basta informar outros 2 parâmetros para o método `view_cell()`, onde o terceiro parâmetro será o tempo em segundos em que o cache deverá ser mantido e o quarto parâmetro é o nome da view no cache, sendo este opcional.

```
view_cell('\App\Libraries\Blog::postsRecentes', 'categoria=tecnologia.
max_posts = 3', 500, 'nome_da_view_cell_no_cache);
```

Capítulo 25
View Parser

O View Parser pode realizar uma substituição simples de texto por pseudovariáveis existentes nas views, podendo analisar e substituir variáveis simples ou pares de tags variáveis.

Para que o View Parser seja capaz de realizar a substituição é necessário que o termo a ser substituído esteja entre chaves, como no exemplo a seguir:

```
<html>
<head>
    <title>{titulo_da_pagina}</title>
</head>
<body>
        <h1>{mensagem}</h1>

        {conteudos}
            <h5>{titulo}</h5>
            <p>{texto}</p>
        {/conteudos}

</body>
</html>
```

As variáveis utilizadas no View Parser não são variáveis reais do PHP, mas representações de texto simples capazes de eliminar o código PHP das views.

O uso de View Parser não é obrigatório, até porque o PHP puro utilizado nas views com um *view render* permite que ele funcione um pouco mais rápido. Mas, em alguns casos, por exemplo, quando se trabalha em conjunto com um designer que não tem conhecimentos sobre PHP, utilizar as pseudovariáveis é uma ótima forma de ter um trabalho fluido e otimizado.

25.1 Utilizando a classe View Parser

A forma mais simples de utilizar View Parser é como um serviço, carregando-o da seguinte forma:

```
$parser = \Config\Services::parser();
```

Outra maneira de utilizar View Parser, caso ela não seja o renderizador padrão, é instanciando-a de forma direta:

```
$parser = new \CodeIgniter\View\Parser();
```

Feito isso você poderá utilizar qualquer um dos 3 métodos de rederização padrão que View Parser fornece:

- `render(viewpath, options, save)`
- `setVar(name,value, context)`
- `setData(data, context)`

Você também poderá especificar delimitadores diretamente através do método `setDelimiter(left, right)` .

Ao utilizar o View Parser, suas views são processadas apenas pelo próprio Parser e não como um script PHP de visualização convencional. O código PHP existente na view é ignorado pelo analisador e apenas as substituições são executadas.

O que View Parser faz de fato

A classe `Parser` processa scripts PHP/HTML armazenados no diretório das views, e esses scripts não podem conter nenhum código PHP.

Cada pseudovariável aciona uma substituição baseada no tipo de valor que foi fornecida para ela. As pseudovariáveis não são extraídas para variáveis PHP, seu valor é acessado através da

sintaxe da pseudovariável, onde seu nome é mencionado entre chaves.

A classe `Parser` faz uso de um array associativo interno para acumular configurações de pseudovariáveis até que o método `render()` seja chamado, significando que os nomes das pseudovariáveis precisam ser exclusivos ou uma configuração de parâmetro posterior substituirá uma anterior.

Esse processo também afeta os valores de parâmetros de escape para contextos diferentes dentro do script, sendo necessário atribuir a cada valor de escape um nome de parâmetro exclusivo.

25.2 Templates

Você pode utilizar o método `render()` para analisar ou renderizar suas views. Altere o método `index()` arquivo `/app/Controllers/Blog.php` de acordo com o código abaixo para que ele passe a utilizar o parser.

```
public function index()
{
    $parser = \Config\Services::parser();
    $dados = [
        "titulo_da_pagina" => "Livro CodeIgniter 4",
        "mensagem"         => "Minha primeira view com o CodeIgniter 4"
    ];
    echo $parser->setData($dados)->render('Cabecalho');
    echo $parser->setData($dados)->render('Blog/Index');
    echo view('Rodape', [], ['cache' => 60, 'cache_name' =>
'rodape_cache']);
}
```

Os parâmetros de exibição que serão substituídos nas pseudovariáveis foram definidos na matriz `$dados` e informados ao Parser por `setData()`. Após definidos os dados, é chamado o

método `render()` que recebe como parâmetro o nome da `view` que deverá ser renderizada.

Agora substitua nas views `/app/Views/Cabecalho.php` e `/app/Views/Blog/Index.php` as chamadas usando PHP pelas pseudovariáveis `{titulo_da_pagina}` e `{mensagem}`.

Opções de configurações do Parser

Veja a seguir uma lista de possíveis configurações que podem ser passadas para os métodos `render()` e `renderString()`.

- `cache` : o tempo em segundos para manter os resultados de uma view em cache (ignorado por `renderString()`);
- `cache_name` : o ID utilizado para salvar/recuperar os resultados de uma view armazenados em cache cujo padrão é o caminho da view (ignorado para `renderString()`);
- `saveData` : `true` se os parâmetros de visualização dos dados precisarem ser armazenados para chamadas futuras (o padrão é `false`);
- `cascadeData` : `true` se as configurações de pseudovariáveis devem ser passadas para substituições aninhadas (o padrão é `true`).

```
echo $parser->render('view', [
    'cache'      => HOUR,
    'cache_name' => 'valor_unico',
]);
```

25.3 Substituindo pseudovariáveis

O CodeIgniter 4 suporta três tipos de substituições: *simples*, *em loop* e *aninhadas*. Essas substituições acontecem na mesma sequência em que as pseudovariáveis foram inseridas na view.

O `Parser` é capaz de levar a substituição muito mais além através dos "pares de variáveis", utilizados para substituições aninhadas ou em loop, e com algumas construções avançadas para substituição condicional.

Quando o Parser é executado, em geral, ele:

- lida com quaisquer substituições condicionais;
- manipula quaisquer substituições aninhadas/em loop;
- lida com as substituições únicas restantes.

Substituição simples

É uma substituição individual de pseudovariáveis em que o parâmetro de dados correspondente possui um valor escalar ou string como no exemplo a seguir:

```
$template = '<head><title>{titulo_blog}</title></head>';
$dados = ['titulo_blog' => 'Blog Pessoal'];

echo $parser->setData($dados)->renderString($template);

// Resultado da renderização: <head><title>Blog Pessoal</title></head>
```

Substituição em loop

Uma substituição em loop acontece sempre que o valor de uma pseudovariável é um array multidimensional.

No primeiro exemplo deste capítulo foi apresentada uma estrutura de view que faz uso da substituição em loop. A seguir você pode ver como o array com os dados deve ser construído para tornar a execução do loop possível durante a renderização.

```
$dados = [
    'titulo_da_pagina' => 'Livro CodeIgniter 4',
    'mensagem'         => 'Minha primeira view com o CodeIgniter 4',
    'conteudos'    => [
            ['titulo' => 'Conteúdo #1', 'texto' => 'Texto para o conteúdo
```

```
#1'],
            ['titulo' => 'Conteúdo #2', 'texto' => 'Texto para o conteúdo
#2'],
            ['titulo' => 'Conteúdo #3', 'texto' => 'Texto para o conteúdo
#3']
    ]
];
```

A seguir, o código HTML da view `/app/Views/Blog/Index.php` onde as pseudovariáveis serão substituídas, inclusive com a execução da substituição em loop.

```
<h1>{mensagem}</h1>

{conteudos}
    <h2>{titulo}</h2>
    <p>{texto}</p>
    <hr/>
{/conteudos}
```

O valor para a pseudovariável `{conteudos}` é multimensional onde o nível externo não possui chaves associadas a cada uma das linhas aninhadas.

Se seus dados de par vierem de um resultado de banco de dados, que já é um array multidimensional, você pode simplesmente usar o método `getResultArray()` para obter o array a ser utilizado no loop.

```
$SQL = $db->query("SELECT * FROM blog_posts");

$dados = [
        'titulo_da_pagina' => 'Livro CodeIgniter 4',
        'mensagem'         => 'Minha primeira view com o CodeIgniter 4',
        'conteudos'        => $SQL->getResultArray()
];

echo $parser->setData($dados)->render('view');
```

Se o array cujo loop você está tentando executar contiver objetos em vez de arrays, o Parser primeiro procurará um método `asArray` no objeto e, se existir, ele será chamado e o array resultante será

repetido como descrito anteriormente. Caso não exista um método `asArray`, o objeto será convertido como um array mantendo suas propriedades públicas e protegidas (exceto a propriedade `_options`) e as disponibilizará para o Parser.

Substituições aninhadas

Uma substituição aninhada acontece quando o valor de uma pseudovariável é um array de valores, e não um array de array. Assim, em vez de fazer um loop repetindo o conteúdo, ela renderiza o bloco de conteúdo cujas pseudovariáveis correspondem às chaves do array passado como valor.

Essa é uma forma muito útil de organizar as informações a serem exibidas caso esteja usando View Parser para reduzir a quantidade de código PHP nas views.

Veja no exemplo a seguir que no array de dados temos as informações a serem enviadas para view e tanto `informacoes` quanto `conteudos` recebem arrays. A diferença é que `informações` recebe um array associativo, enquanto `conteudos` é um array multidimensional, contendo outros arrays.

```
$dados = [
    'titulo_da_pagina' => 'Livro CodeIgniter 4',
    'mensagem'         => 'Minha primeira view com o CodeIgniter 4',
    'informacao'       => [
        'titulo'    => 'Título da Informação',
        'descricao' => 'Descrição da informação'
    ],
    'conteudos'   => [
        ['titulo' => 'Conteúdo #1', 'texto' => 'Texto para o conteúdo
#1'],
        ['titulo' => 'Conteúdo #2', 'texto' => 'Texto para o conteúdo
#2'],
        ['titulo' => 'Conteúdo #3', 'texto' => 'Texto para o conteúdo #3']
    ]
];
```

```
echo $parser->setData($dados)->render('view');
```

O valor para a pseudovariável `{informacao}` é um array associativo. Os pares de chave/valor dentro dele serão expostos dentro do loop de pares de variáveis para essa variável.

Veja a seguir um exemplo de estrutura HTML para a view `/app/Views/Blog/Index.php` funcionar com essa estrutura de dados:

```
<h1>{mensagem}</h1>

{informacao}
<code>
    <h3>{chamada}</h3>
    <p>{descricao}</p>
</code>
{/informacao}

{conteudos}
<h2>{titulo}</h2>
<p>{texto}</p>
<hr />
{/conteudos}
```

Acessando pseudovariáveis externas dentro de um loop

Pode acontecer de sua aplicação precisar exibir informações dentro de um loop ou dados aninhados, sendo que os valores das pseudovariáveis estão fora do array a ser utilizada. Para que isso seja possível, você precisa definir a opção `cascadeData` como `true`, assim, sempre que precisar exibir uma pseudovariável que não faz parte do array dentro do loop ou aninhamento, isso será possível.

```
$dados = [
    'titulo_da_pagina' => 'Livro CodeIgniter 4',
    'mensagem'         => 'Minha primeira view com o CodeIgniter 4',
    'informacao'       => [
        'titulo'    => 'Título da Informação',
        'descricao' => 'Descrição da informação'
```

```
    ],
    'conteudos'   => [
        ['titulo' => 'Conteúdo #1', 'texto' => 'Texto para o conteúdo
#1'],
        ['titulo' => 'Conteúdo #2', 'texto' => 'Texto para o conteúdo
#2'],
        ['titulo' => 'Conteúdo #3', 'texto' => 'Texto para o conteúdo #3']
    ]
];

echo $parser->setData($dados)->render('view', ['cascadeData' => true]);
```

```
<h1>{mensagem}</h1>

{conteudos}
<h2>{titulo} | {titulo_da_pagina}</h2>
<p>{texto}</p>
<hr />
{/conteudos}
```

Se você não informar todas as chaves no array para substituir as pseudovariáveis na sua view, a renderização vai imprimir na tela a própria pseudovariável, então certifique-se de que todas as variáveis foram devidamente definidas.

25.4 Aplicando lógica condicional com pseudovariáveis

A classe `Parser` suporta alguns tipos de lógica condicional mais básicos para manipular a sintaxe `if`, `else` e `elseif`, sendo que todos os blocos devem ser fechados com a tag `endif`.

Usando PHP

```
<?php if ($mensagem != ""): ?>
    <h1>Imprime Conteúdo</h1>
<?php else: ?>
```

```
    <h1>Sem Mensagem</h1>
<?php endif; ?>
```

Usando pseudovariáveis

```
{if $mensagem != ""}
    <h1>{mensagem}</h1>
{else}
    <h1>Sem Mensagem</h1>
{endif}
```

25.5 Filtros

Os filtros fornecem maneiras de reutilizar os mesmos dados com apresentações diferentes e podem ser aplicados a qualquer substituição de variável única com o objetivo de modificar a maneira como a variável é apresentada.

Um exemplo muito comum de formatação de dados em que os filtros se encaixam é quando há a necessidade de trabalhar com datas. Pode acontecer de a mesma data ser exibida com formatações diferentes dentro a mesma página, e os filtros são muito úteis nesse caso.

Eles são comandos que vêm logo após o nome da pseudovariável e são separados pelo símbolo `|` :

```
{criado_em| date(d/m/Y)}
```

É possível aplicar múltiplos filtros a uma mesma pseudovariável e eles serão executados em ordem, da esquerda para a direita:

```
{criado_em| date_modify(+5 days)| date(d/m/Y)}
```

Veja a seguir uma lista com alguns dos filtros que estão disponíveis para uso no Parser. A lista completa pode ser vista na documentação oficial do CodeIgniter 4.

- `abs` : exibe o valor absoluto de um número
  - **Parâmetro(s):** nenhum
  - `{v| abs}`
- `date` : formatação de datas compatível com os padrões do PHP
  - **Parâmetro(s):** formato da data `(d/m/Y)`
  - `{v| date(d/m/Y)}`
- `date_modify` : string `strtotime` compatível para modificar datas (+5 days, -1 week, ...)
  - **Parâmetros:** valor a ser adicionado ou subtraído (+5 days, -1 day, +1 week, ...)
  - `{v| date_modify(+1 day)`
- `limit_chars` : limita o número de caracteres para uma string
  - **Parâmetro:** número inteiro que representa o limite de caracteres
  - `{v| limit_chars(144)}`
- `limit_words` : limita o número de palavra para uma string
  - **Parâmetro:** número inteiro que representa o limite de palavras
  - `{v| limit_chars(144)}`

Veja outros filtros em https://codeigniter.com/user_guide/outgoing/view_parser.html#provided-filters

Filtros customizados e funções nativas do PHP como filtros

Filtros customizados podem ser criados facilmente no CodeIgniter 4. Para isso basta editar o arquivo `/app/Config/View.php` e adicionar os filtros ao array `$filters` .

Cada chave do array corresponde ao nome do filtro a ser chamado na visualização e o valor pode ser qualquer código PHP válido que possa ser chamado.

```
public $filters = [
    'abs'        => '\CodeIgniter\View\Filters::abs',
```

```
    'capitalize' => '\CodeIgniter\View\Filters::capitalize',
];
```

Também é possível utilizar funções nativas do PHP como filtros, e a forma de se fazer isso é semelhante à apresentada anteriormente, com a diferença de que, em vez de associar um `Namespace` ao array, será associada à função nativa do PHP.

```
public $filter = [
    'str_repeat' => '\str_repeat'
]
```

Capítulo 26
Models

O model é onde fica a parte lógica da sua aplicação, digamos que é a inteligência dela. Distribuindo as inteligências da aplicação pelo padrão MVC podemos ter algo semelhante ao seguinte:

- **Model:** sabe o que fazer
- **View:** sabe o que responder ao usuário
- **Controller:** sabe quando fazer, ou seja, ele usa a inteligência do model com a habilidade de apresentação da view para completar o ciclo do MVC

O CodeIgniter fornece uma variedade de recursos bastante interessantes para otimizar o seu trabalho e torná-lo cada vez mais ágil, de modo que você possa focar na inteligência da sua aplicação e não em ficar *reinventando a roda*.

- conexão automática com banco de dados
- métodos CRUD básicos já construídos
- validação de dados direto no **model**
- paginação automática de resultados

26.1 Criando um model

Não é necessário estender nenhuma classe especial para criar models em suas aplicações, tudo o que você precisa é de uma instância da conexão com o banco de dados.

Os models são criados dentro do diretório `/app/Models`, ou dentro do diretório do módulo do qual ele faz parte caso sua aplicação esteja modularizada. Veja a seguir o exemplo da estrutura inicial de um

model com a instância da conexão com o banco de dados criada no método `__construct()`:

```php
<?php namespace App\Models;

    use CodeIgniter\Database\ConnectionInterface;

    class PostModel
    {
        protected $db;

        public function __construct(ConnectionInterface &$db)
        {
            $this->db =& $db;
        }
    }
```

Caso você não instancie a conexão com o banco de dados, ela será feita com o grupo padrão conforme estiver definido nas configurações em `/app/Config/Database.php` ou em `/.env`.

É possível modificar esse grupo adicionando a propriedade `$DBGroup` à classe, garantindo assim que dentro do model, qualquer referência feita a `$this->db` utilizará a conexão com o grupo especificado em `$DBGroup`.

```php
<?php namespace App\Models;

    use CodeIgniter\Database\ConnectionInterface;

    class PostModel
    {
        protected $db;
        protected $DBGroup = 'blog';

        public function __construct(ConnectionInterface &$db)
        {
            $this->db =& $db;
        }
    }
```

26.2 Configurando o model

O model no CodeIgniter 4 possui opções de configuração capazes de ajudar bastante no processo de desenvolvimento de aplicações e no exemplo a seguir você pode ver essas configurações aplicadas ao model.

```
<?php namespace App\Models;

    use CodeIgniter\Database\ConnectionInterface;

    class PostModel
    {
        protected $table      = 'posts';
        protected $primaryKey = 'id';

        protected $returnType = 'array';
        protected $useSoftDeletes = true;

        protected $allowedFields = ['titulo', 'descricao', 'conteudo'];

        protected $useTimestamps = false;
        protected $createdField  = 'created_at';
        protected $updatedField  = 'updated_at';
        protected $deletedField  = 'deleted_at';

        protected $dateFormat         = date('Y-m-d H:i:s')
        protected $validationRules    = [];
        protected $validationMessages = [];
        protected $skipValidation     = false;
    }
```

- `$table` : especifica a tabela onde o model executará as operações.
- `$primaryKey` : especifica qual a coluna no banco de dados está definida como chave primária.
- `$returnType` : especifica o formato dos dados que serão retornados pelos métodos executados no model. Aceita os valores `array` , `object` ou o nome completo de uma classe que

pode ser utilizada como o método `getCustomResultObject()` do objeto `Result`.

- `$useSoftDeletes`: se o valor informado for `true` sempre que o método `delete` for executado, em vez de excluir o registro será adicionada uma informação de exclusão na coluna `delete_at` na tabela do banco de dados, ou em outra que tenha sido devidamente configurada em `$deletedField`.

Para utilizar `$useSoftDeletes` é necessário que haja uma coluna na tabela para receber essa informação, que deve ser do tipo `datetime` ou `integer`, de acordo com a configuração de `$dateFormat` no model. Por padrão, o nome da coluna é `delete_at`, mas ele pode ser alterado para qualquer outro nome de sua escolha, desde que seja especificado em `$deleteField` nas configurações do model.

- `$allowedFields`: esse array contém os campos que podem ser definidos ao inserir ou atualizar registros. Nomes de campos diferentes dos definidos no array serão descartados.
- `$useTimestamps`: determina se a data atual deve ser adicionada de forma automática a todas as inserções e atualizações de registros. Se definido como `true`, definirá a hora atual no formato especificado em `$dateFormat`.

Para fazer uso desse recurso é necessário que a tabela no banco de dados tenha as colunas `created_at` e `updated_at` ou então com nomes de sua escolha, desde que informados nas configurações do model em `$createdField` e `$updatedField`.

- `$createdField`: responsável por identificar qual o campo deve receber as informações de criação do registro.
- `$updatedField`: responsável por identificar qual o campo deve receber as informações de atualização do registro.

- `$updatedField` : responsável por identificar qual o campo deve receber as informações de remoção do registro.
- `$dateFormat` : por padrão insere valores `DATETIME` , mas pode utilizar `date` ou `int` (para o caso de usar timestamp).
- `$validationRules` : contém um array com regras de validação ou uma sequência que contém o nome do grupo de validação.
- `$validationMessages` : contém um array de mensagens de erro personalizadas que devem ser utilizadas no processo de validação.
- `$skipValidation` : define se a validação deve ou não ser ignorada nos processos de inserção e atualização. O valor padrão é `false` , ou seja, sempre haverá tentativa de validação dos dados.
- `$beforeInsert` , `$afterInsert` , `$beforeUpdate` , `$afterUpdate` , `$afterFind` , `$afterDelete` : arrays que permitem especificar *callbacks* a serem executados após as execuções antes e/ou depois de inserções, atualizações, pesquisas e exclusões de registros.

26.3 Inserir, atualizar, excluir e localizar registros

Essas são as 4 operações de base que toda aplicação precisa executar para ter um funcionamento perfeito. O CodeIgniter ajuda muito no processo de construção dessas rotinas pois ele possui um conjunto de métodos auxiliares que podem ser utilizados livremente dentro do model.

A seguir, você verá alguns dos principais métodos que pode utilizar em suas aplicações para obter melhor desempenho tanto no funcionamento quanto no desenvolvimento sem ter que escrever código adicional, usando apenas o código nativo.

Métodos para inserir e atualizar registros

insert()

Esse método é o responsável por inserir um registro no banco a partir de um array associativo passado como único parâmetro. As chaves do array passado como parâmetro devem corresponder aos nomes das colunas na tabela.

```
$dados = [
    'titulo'    => 'Livro CodeIgniter 4',
    'descricao' => 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc blandit auctor ornare.',
    'conteudo'  => 'Sed porttitor ipsum vitae imperdiet pellentesque. Quisque id suscipit tellus. Vivamus eu ultricies dolor...'
];

$postModel->insert($dados);
```

$update()

Atualiza os dados de um registro existente no banco de dados. O primeiro parâmetro é a `$primaryKey` do registro a ser atualizado. O segundo parâmetro é um array associativo como o utilizado no `insert()`.

```
$dados = [
    'titulo'    => 'Livro CodeIgniter 4',
    'descricao' => 'Praesent laoreet mi ut mauris rhoncus, in luctus neque vestibulum. Ut bibendum ante at ex pellentesque vehicula.',
    'conteudo'  => 'Sed porttitor ipsum vitae imperdiet pellentesque. Quisque id suscipit tellus. Vivamus eu ultricies dolor...'
];

$postModel->update($key, $dados);
```

Também é possível atualizar múltiplos registros através do método `update()`, para isso basta passar como primeiro parâmetro um array com as chaves primárias dos registros a serem atualizados, e todos os especificados serão atualizados com os dados passados no segundo parâmetro.

```
$dados = ['publicado' => 1];
$postModel->update([1,3,5],$dados);
```

Mas também é possível combinar outros métodos, como o `whereIn` e o `set` por exemplo, para flexibilizar o processo.

```
$postModel->whereIn('id', [1,3,5])->set(['publicado' => 1])->update();
```

Dessa forma você ganha flexibilidade para utilizar recursos do Query Builder e mantém os benefícios de validação, eventos, entre outros do próprio model.

Métodos para excluir registros

`delete()`

Exclui um registro de acordo com a chave primária, ou um array contendo diversas chaves primárias, passado como primeiro parâmetro. Se `$useSoftDeletes` estiver definido como `true`, o método `delete` atualizará a coluna `delete_at` do registro com a data e hora atuais, caso contrário, o registro será removido de forma definitiva do banco de dados.

```
$postModel->delete(12);
$postModel->delete([1,3,5]);
```

Caso `$useSoftDeletes` esteja definido como `true` e você queira forçar a remoção definitiva do registro no banco de dados, basta passar `true` como segundo parâmetro para o método `delete()`.

`purgeDelete()`

Remove todos os registros do banco de dados cujo valor para a coluna `delete_at` seja diferente de `null`.

```
$postModel->purgeDelete();
```

Métodos para localização de registros

`find()`

Retorna um único registro de acordo com a chave primária passada como primeiro parâmetro. O formato dos dados retornados é o mesmo definido em `$returnType`.

O parâmetro pode ser passado como um valor único ou como um array. Nesse caso serão retornados os registros de acordo com os valores contidos no array informado.

```
$post = $postModel->find($post_id);
$posts = $postModel->find([1,3,5]);
```

Se nenhum parâmetro for informado, todos os registros da tabela serão retornados.

`findAll()`

Retorna todos os registros da tabela no formato definido em `$returnType`.

```
$posts = $postModel->findAll();
```

A consulta realizada por esse método pode ser modificada utilizando métodos como `where`, `order_by` e outros disponíveis.

```
$posts = $postModel->where('publicado', 1)->findAll();
```

Também é possível paginar os resultados de `findAll()` passando os parâmetros `$limit` e `$offset`, respectivamente o número limite de registros e página.

```
$posts = $postModel->findAll($limit, $offset);
```

`withDeleted()`

Se `$useSoftDeletes` estiver definido como `true`, o método utilizado de forma combinada com `withDeleted()` retornará todos os registros incluindo os marcados como excluídos.

```
// retorna todos os registros, exceto os que foram excluídos
$posts = $postModel->findAll();
```

```php
// retorna todos os registros, excluídos ou não
$posts = $postModel->withDeleted()->findAll();
```

`onlyDeleted()`

Retorna apenas os registros excluídos.

```php
$posts_excluidos = $postModel->onlyDeleted()->findAll();
```

`save()` : o grade curinga

O método `save()` funciona como uma combinação dos métodos `insert()` e `update()`, manipulando de forma automática as inserções de dados no banco. Ele trabalha em conjunto com a chave `$primaryKey` configurada no model, de modo que, se no array de dados passado como parâmetro uma das chaves corresponder ao valor de `$primaryKey`, ele entende que deve executar um `update()`, caso contrário, executa um `insert()`.

```php
// Sem a chave primária definida é executado o insert
$dados = [
    'titulo'    => 'Livro CodeIgniter 4',
    'descricao' => 'Praesent laoreet mi ut mauris rhoncus, in luctus neque vestibulum. Ut bibendum ante at ex pellentesque vehicula.',
    'conteudo'  => 'Sed porttitor ipsum vitae imperdiet pellentesque. Quisque id suscipit tellus. Vivamus eu ultricies dolor...'
];
$postModel->save($dados);

// Com a chave primária definida é executado o update
$dados = [
    'id'        => 1
    'titulo'    => 'Livro CodeIgniter 4',
    'descricao' => 'Praesent laoreet mi ut mauris rhoncus, in luctus neque vestibulum. Ut bibendum ante at ex pellentesque vehicula.',
    'conteudo'  => 'Sed porttitor ipsum vitae imperdiet pellentesque. Quisque id suscipit tellus. Vivamus eu ultricies dolor...'
];
$postModel->save($dados);
```

O método `save()` também é capaz de simplificar o trabalho com objetos de resultados de classe personalizada. Ele reconhece um objeto não simples e captura seus valores públicos e protegidos em um array, que é passado para o método de inserção ou atualização apropriados. Com isso, você pode trabalhar com classes de entidade de uma forma mais limpa.

> Classes de entidades são classes simples que representam uma única instância de um tipo de objeto, como um usuário, um post no blog, um comentário etc. Essa classe é responsável por manter a lógica de negócios em torno do próprio objeto.

```php
namespace App\Entities;

class Post
{
    protected $id;
    protected $titulo;
    protected $descricao;
    protected $conteudo

    public function __get($key)
    {
        if (property_exists($this, $key))
        {
            return $this->$key;
        }
    }

    public function __set($key, $value)
    {
        if (property_exists($this, $key))
        {
            $this->$key = $value;
        }
    }
}
```

Veja a seguir uma estrutura inicial de um model para trabalhar com classe de entidade:

```
use CodeIgniter\Model;

class PostModel extends Model
{
    protected $table = 'posts';
    protected $returnType = '\App\Entities\Post';
    protected $allowedFields = [
        'titulo', 'descricao', 'conteudo'
    ];
}
```

No código apresentado, repare que foi passado como valor para `$returnType` o namespace da classe de entidade `\App\Entities\Post`, em vez de ter sido passado um valor como `object` ou `array`, por exemplo.

Isso faz com que o model trabalhe com os dados da tabela `posts` e retorne os resultados como uma instância de `\App\Entities\Post`. Assim, sempre que você precisar persistir os registros no banco de dados, terá que escrever métodos personalizados ou usar o método `save()` do model para inspecionar a classe, obter propriedades públicas e privadas e salvá-las no banco de dados.

```
// Recupera uma instância de Post
$post = $model->find(1);

// Aplica algumas alterações
$post->titulo = "Livro CodeIgniter 4 - Classe de Entidade";

// Salvando as mudanças
$model->save($post);
```

No próximo capítulo você verá mais detalhes sobre o uso de entidades de classe.

26.4 Validando dados

Validar os dados no model é uma excelente maneira de evitar códigos duplicados. A classe `Model` no CodeIgniter 4 fornece uma forma de validação automática para todos os dados antes que eles sejam armazenados no banco de dados através dos métodos `save()`, `insert()` ou `update()`.

Para colocar essa validação em funcionamento, é preciso configurar as propriedades `$validationRules`, responsável por definir os campos e as regras de validação a serem aplicadas a cada um, e `$validationMessages`, que é responsável pela personalização das mensagens de erro para cada regra de validação definida em `$validationRules`.

```
class PostModel extends Model
{
    protected $validationRules    = [
        'titulo'     => 'required|max_length[15]',
        'descricao'  => 'required|max_length[160]',
        'conteudo'   => 'required|min_length[1000]'
    ];

    protected $validationMessages = [
        'titulo'    => [
            'required'   => 'Informe o título do post'
        ],
        'descricao' => [
            'max_length' => 'O tamanho máximo para a descrição é de 160
caracteres'
        ],
        'conteudo'  => [
            'min_length' => 'O tamanho mínimo para o conteúdo é de 1000
caracteres'
        ]
    ];
}
```

Para recuperar os erros de validação após a execução, você pode utilizar o método `$model->errors()`. Ele retorna um array com os nomes dos campos e seus erros, que podem ser utilizados para exibir os erros de forma individual ou agrupada para o usuário através da view.

Checando o status da operação e enviando os erros para a view

```
if ($model->save($data) === false)
{
    return view('updatePost', ['errors' => $model->errors()]);
}
```

Renderizando erros nas view

```
<?php if (! empty($errors)) : ?>
    <div>
    <?php foreach ($errors as $field => $error) : ?>
            <p><?= $error ?></p>
    <?php endforeach ?>
    </div>
<?php endif ?>
```

Mensagens de erro nos arquivos de idioma

O CodeIgniter possui tradução para todas as suas mensagens através dos arquivos de idioma, mas caso você queira utilizar seu próprio arquivo de idioma para definir as mensagens de erro das validações, basta criá-lo no diretório `/app/Language` colocando-o dentro do diretório do idioma desejado.

> Para saber como criar arquivos de tradução, acesse o capítulo **Criando arquivos de tradução**.

Na hora de definir as mensagens de erro na validação dentro do Model, basta você fazer a chamada para o arquivo de tradução usando a sintaxe dos pontos, como no exemplo a seguir:

```php
class PostModel extends Model
{
    protected $validationRules    = [
        'titulo'     => 'required| max_length[15]',
        'descricao'  => 'required| max_length[160]',
        'conteudo'   => 'required| min_length[1000]'
    ];

    protected $validationMessages = [
        'titulo'    => [
            'required'   => 'Regras.titulo.required'
        ],
        'descricao' => [
            'max_length' => 'Regras.descricao.max_length'
        ],
        'conteudo'  => [
            'min_length' => 'Regras.conteudo.min_length'
        ]
    ];
}
```

Capítulo 27
Entidades de Classe

O CodeIgniter 4 traz o suporte a entidades de classe que até então não existiam no framework, o que lhe dá mais uma opção para trabalhar com os dados no model de forma organizada e limpa.

Simplificando, uma entidade de classe é simplesmente uma classe que representa um único registro do banco de dados. Possui propriedades de classe para representar colunas do banco de dados e fornece métodos adicionais para implementar regras de negócios para esse registro.

A entidade de classe não sabe como realizar a persistência de dados, então ela precisa trabalhar em conjunto com o model para que isso seja possível. Assim, se algo mudar na forma de salvar o objeto não será necessário alterar a forma como esse objeto é utilizado em toda a aplicação.

Veja a seguir uma entidade de classe mais simples e como você pode fazer com que ela fique mais robusta:

```
id         - integer
nome       - string
email      - string
senha      - string
biografia  - string
status     - boolean
created_at - datetime
```

27.1 Criando uma entidade de classe

Não existe um diretório específico para armazenar os arquivos das entidades de classe que serão utilizadas pela aplicação, então

sugiro a criação de um diretório `Entities` dentro de `/app`. Em seguida, crie um arquivo dentro desse diretório chamado `Autor.php` e adicione a estrutura a seguir:

```
<?php namespace App\Entities;

use CodeIgniter\Entity;

class Autor extends Entity
{

}
```

Com a estrutura da entidade de classe criada, você deve criar o model que fará uso dela. Então crie um model chamado `AutorModel.php` dentro do diretório `/app/Models` e adicione a ele a estrutura a seguir:

```
<?php namespace App\Models;

use CodeIgniter\Model;

class AutorModel extends Model
{
    protected $table         = 'autores';
    protected $allowedFields = ['nome', 'email', 'senha', 'biografia',
'status'];
    protected $returnType    = 'App\Entities\Autor';
    protected $useTimestamps = true;
}
```

Esse model se comunica com a tabela `autores` no banco de dados, e isso foi definido através da propriedade `$table`. Os campos que podem ser alterados por classes externas foram definidos em `$allowedFields`, e a definição automática de datas de criação, alteração e exclusão foi ativada na propriedade `$useTimestamps`.

Ao definir `$returnType` como uma entidade de classe, você garante que todos os métodos no model que se comunicam com o banco de dados e obtêm registros retornem instâncias da classe `Autor` em

vez de um objeto ou array, como de costume nas versões anteriores do CodeIgniter.

27.2 Trabalhando com uma entidade de classe

Após criar a entidade de classe e o model que fará uso dela você já tem em mãos a base que precisa para começar o uso desse recurso em sua aplicação. E o processo é o mesmo como o realizado em qualquer outra classe.

Se você analisar bem, a classe `Autor` não tem nenhuma propriedade definida para as colunas de sua respectiva tabela no banco de dados, mas é possível acessá-las como se fossem propriedades públicas. A classe base `CodeigniterEntity` cuida disso de forma automática, e também dá a capacidade de verificar as propriedades com `isset()` ou `unset()`, além de acompanhar quais colunas foram alteradas desde o momento em que o objeto foi criado ou recuperado através de consulta ao banco de dados.

Quando as informações do autor são passadas para o método `save()` no model, ele automaticamente lê as propriedades e salva as alterações nas colunas definidas na propriedade `$allowedFields`. Ele também sabe se deve criar um novo registro ou atualizar um registro existente.

```
$autor = $autorModel->find($id);

// Exibe
echo $autor->nome;
echo $autor->biografia;

// Atualiza
unset($autor->nome);
if (! isset($autor->nome)
{
    $autor->nome = 'Jonathan Lamim';
```

```
}
$autorModel->save($autor);

// Cria
$autor            = new App\Entities\autor();
$autor->nome      = 'Jonathan';
$autor->email     = 'contato@jonathanlamim.com.br';
$autor->senha     = 'minhasenhasemcriptografia'
$autor->biografia = 'Software Developer';
$autor->status    = 1;
$autorModel->save($autor);
```

27.3 Alimentando as propriedades da classe

A classe `Entity` possui um método chamado `fill()`, que é capaz de inserir um array chave/valor na classe e preenchê-la com as devidas propriedades.

Qualquer propriedade desse array será definida na entidade, no entanto, ao salvar os registros, apenas os campos listados em `$allowedFields` terão seu conteúdo armazenado no banco de dados.

```
$dados = $this->request->getPost();

$autor = new App\Entities\autor();
$autor->fill($dados);
$autorModel->save($autor);
```

Também é possível enviar os dados no método construtor e eles serão transmitidos pelo método `fill()` durante a instanciação.

27.4 Manipulando regras de negócios

A classe base `Entity` implementa alguns métodos como `__get()` e `__set()` que verificam métodos especiais e os utilizam ao invés de usar os atributos de forma direta, permitindo impor qualquer regra de negócio ou conversão de dados necessária à aplicação.

Veja a seguir um exemplo de entidade com a implementação de algumas regras de negócio:

```php
<?php namespace App\Entities;

use CodeIgniter\Entity;
use CodeIgniter\I18n\Time;

class Autor extends Entity
{
    // A lógica aplicada aqui garante que a senha seja sempre um hash
    public function setSenha(string $senha)
    {
        $this->senha = password_hash($senha, PASSWORD_BCRYPT);
        return $this;
    }

    // Aqui se aplica uma lógica para converter a string correspondente à data
    // em um objeto DateTime, garantindo que a informação seja armazenada corretamente
    public function setCreatedAt(string $dateString)
    {
        $this->attributes['created_at'] = new Time($dateString, 'UTC');
        return $this;
    }

    // Aqui é feita a conversão dos dados para o horário e formato padrão
    public function getCreatedAt(string $format = 'd-m-Y H:i:s')
    {
        $this->attributes['created_at'] = $this->mutateDate($this->attributes['created_at']);
        $timezone = $this->timezone ?? app_timezone();
        $this->attributes['created_at']->setTimezone($timezone);
        return $this->attributes['created_at']->format($format);
```

```
    }
}
```

A primeira observação a ser feita no código apresentado é o nome dos métodos que foram adicionados. Para cada um, a classe espera que o nome da coluna seja convertido para PascalCase e prefixado com *get* ou *set*.

Esses métodos serão chamados de forma automática sempre que uma propriedade da classe for definida ou recuperada através da sintaxe direta (`$autor->created_at`).

> **ATENÇÃO**
>
> A aplicação automática das regras de negócio para os métodos `setX()` , onde X é o nome da coluna no banco de dados só funciona quando as propriedades forem acessadas de fora da classe `Entity` . Métodos internos que precisem acessar as propriedades devem chamá-las diretamente pelo nome.

Usar a classe `Entity` vai dar a você mais flexibilidade na construção das regras de negócio, mantendo sua aplicação organizada e permitindo maior produtividade e performance no processo de desenvolvimento.

27.5 Mapeamento de dados

O mapeamento de dados é uma forma eficaz para que os nomes das colunas existentes no banco de dados sejam mapeados e, em caso de alteração na estrutura do banco, você não precise alterar todas essas informações ao longo do código da sua aplicação.

Imagine o cenário onde você desenvolveu uma aplicação para seu cliente e algum tempo depois ele quer fazer um upgrade nela.

Durante o processo de atualização você percebe que além de precisar atualizar a estrutura do banco de dados com novas colunas e tabelas, precisa alterar o nome de algumas colunas para fazerem mais sentido dentro da aplicação.

Essas mudanças em nomes de colunas que estão em uso são capazes de gerar grande trabalho, pois você precisa alterar todas as ocorrências desses nomes na aplicação. Usando o mapeamento de dados na classe `Entity` você não precisará fazer isso, apenas remapeá-las, mantendo os mesmos nomes já utilizados.

Versão 1.0 da aplicação

```php
<?php namespace App\Entities;

use CodeIgniter\Entity;

class Autor extends Entity
{
    protected $attributes = [
        'id'         => null,
        'nome'       => null,
        'email'      => null,
        'senha'      => null,
        'biografia'  => null,
        'status'     => null,
        'created_at' => null,
        'updated_at' => null,
    ];
}
```

Versão 2.0 da aplicação

O campo `status` deve passar a se chamar `ativo`.

```php
<?php namespace App\Entities;

use CodeIgniter\Entity;

class Autor extends Entity
```

```
{
    protected $attributes = [
        'id'         => null,
        'nome'       => null,
        'email'      => null,
        'senha'      => null,
        'biografia'  => null,
        'status'     => null,
        'created_at' => null,
        'updated_at' => null,
    ];

    protected $datamap = [
        'ativo' => 'status'
    ];
}
```

Ao adicionar o array `$datamap` e nele associar como chave o nome da coluna no banco de dados e como valor a propriedade da classe, você está dizendo à classe em qual propriedade a coluna do banco de dados deve estar acessível.

Dessa forma, sua aplicação poderá continuar utilizando a propriedade `$autor->status` que o mapeamento se encarregará de ajustar os nomes no momento em que esses dados forem enviados ou recuperados no banco de dados.

Verificando a mudança de atributos

A classe `Entity` permite verificar se um atributo foi modificado ou não, recurso muito útil em processos de atualização de dados e validações. Basta utilizar o método `hasChanged()` passando como parâmetro o nome do campo a ser verificado:

```
$autor = new Autor();
$autor->hasChanged('nome'); //retornará false

$autor->nome = 'Lamim';
$autor->hasChanged('nome'); //retornará true
```

Caso você queira verificar mudanças em toda a entidade, basta chamar o método `hasChanged()` sem informar o parâmetro, e caso exista alguma alteração, o retorno será `true`, caso não tenha havido alterações, será `false`.

27.6 Conversão de dados

Datas

Os campos `created_at`, `updated_at` e `delete_at` serão convertidos em instâncias `Time` sempre que forem inseridos ou recuperados. O fato de serem convertidos para instâncias `Time` dá a você de forma automática métodos que podem ser utilizados diretamente na propriedade ao manipulá-la.

As propriedades a serem convertidas de forma automática devem ser informadas no array `$dates`:

```
<?php namespace App\Entities;

use CodeIgniter\Entity;

class Autor extends Entity
{
    protected $dates = ['created_at', 'updated_at', 'deleted_at'];
}
```

Um outro detalhe nessa conversão é que os parâmetros utilizados na formatação da data serão os mesmos da configuração da aplicação, feita em `/app/Config/App.php` no que diz respeito ao timezone.

```
$autor = new App\Entities\Autor();

// Convertendo para Instância Time
$autor->created_at = '2019-11-07 08:22:32';
```

```php
// Alguns dos métodos da instância Time que podem ser utilizados
echo $autor->created_at->humanize();
echo $autor->created_at->setTimezone('America/Sao_Paulo')->toDateString();
```

Tipos de dados

É possível especificar que as propriedades na entidade sejam convertidas em tipos de dados comuns com a propriedade de conversão. Para isso devem ser listadas em um array chamado `$casts`, onde a chave é o nome da propriedade da classe e o valor é o tipo de dados para o qual deve ser convertido. Essa conversão é aplicada apenas quando os valores são lidos, mantendo-os íntegros no banco de dados.

As propriedades podem ser convertidas para qualquer um dos seguintes tipos de dados:

- integer
- float
- double
- string
- boolean
- object
- array
- datetime
- timestamp

```php
<?php namespace App\Entities;

use CodeIgniter\Entity;

class Autor extends Entity
{
    protected $casts = [
        'status' => 'boolean'
    ],
}
```

Banco de dados

Trabalhar com bancos de dados no CodeIgniter é uma tarefa bastante tranquila, pois ele fornece diversos recursos para otimizar o desenvolvimento dos códigos relacionados ao consumo de dados na aplicação.

Através de recursos nativos e possibilidade de combinar esses recursos, você poderá desenvolver soluções cada vez melhores para as necessidades de cada projeto sem ter de utilizar diversas bibliotecas de terceiros.

Nos capítulos que compõem esta parte do livro você fará uma verdadeira imersão no assunto de banco de dados dentro do CodeIgniter 4.

Capítulo 28
Configurações iniciais

Em toda aplicação em CodeIgniter 4, a primeira coisa a se fazer é aplicar as configurações no framework de acordo com as necessidades da aplicação que será desenvolvida. E nesse processo está a configuração do banco de dados, caso sua aplicação o utilize.

No CodeIgniter 4, você pode aplicar as configurações do banco de dados a partir do arquivo padrão de configuração `/app/Config/Database.php` ou então através de um arquivo `.env` localizado na raiz da aplicação.

28.1 Configurando através do arquivo padrão

Ao utilizar o arquivo de configuração padrão, você trabalhará com um array contendo todos os parâmetros necessários para configurar

a conexão com o banco de dados.

> **INFORMAÇÃO**
>
> O nome `default`, utilizado na configuração inicial do arquivo `/app/Config/Database.php`, não é um padrão obrigatório. Você pode nomear as propriedades que identificam grupos da forma como for mais legível para você.

```
class Database {
    public $default = [
        'DSN'      => '',
        'hostname' => 'localhost',
        'username' => '',
        'password' => '',
        'database' => '',
        'DBDriver' => 'MySQLi',
        'DBPrefix' => '',
        'pConnect' => false,
        'DBDebug'  => (ENVIRONMENT !== 'production'),
        'cacheOn'  => false,
        'cacheDir' => '',
        'charset'  => 'utf8',
        'DBCollat' => 'utf8_general_ci',
        'swapPre'  => '',
        'encrypt'  => false,
        'compress' => false,
        'strictOn' => false,
        'failover' => [],
        'port'     => 3306,
    ];
}
```

Esse array é armazenado em uma propriedade da classe `Database` que identifica o grupo de configuração de acesso ao banco de dados, permitindo que você crie diversos grupos que representem os diferentes ambientes da aplicação (desenvolvimento, testes, produção ou outros).

Com esses diversos grupos configurados você pode alternar entre eles de acordo com o ambiente no momento em que as configurações forem carregadas.

> **DICA**
>
> Você pode utilizar a variável de ambiente `ENVIRONMENT` para definir qual grupo deve ser utilizado. Para isso, utilize como nome do grupo os mesmos nomes atribuídos à variável de ambiente `ENVIRONMENT` , assim você não precisa ficar criando lógica condicional para determinar o grupo a ser utilizado.

```
class Database
{
    public $desenvolvimento = [...];
    public $teste           = [...];
    public $producao        = [...];

    public function __construct()
    {
        $this->defaultGroup = ENVIRONMENT;
    }
}
```

Ao utilizar o arquivo de configuração padrão, você tem a possibilidade de configurar `failovers` que permitem a tentativa de conexão com um banco de dados diferente caso o principal falhe. Para isso, basta você adicionar uma chave chamada `failover` ao array correspondente ao grupo de conexão para o qual você deseja ter essa opção alternativa e passar para ele as conexões alternativas usando a mesma base de informações.

```
$default['failover'] = [
    [
        'hostname' => 'alternativa',
        'username' => '',
        'password' => '',
        'database' => '',
```

```
            'DBDriver' => 'MySQLi',
            'DBPrefix' => '',
            'pConnect' => TRUE,
            'DBDebug'  => TRUE,
            'cacheOn'  => FALSE,
            'cacheDir' => '',
            'charset'  => 'utf8',
            'DBCollat' => 'utf8_general_ci',
            'swapPre'  => '',
            'encrypt'  => FALSE,
            'compress' => FALSE,
            'strictOn' => FALSE
        ],
        [
            'hostname' => 'alternativa02',
            'username' => '',
            'password' => '',
            'database' => '',
            'DBDriver' => 'MySQLi',
            'DBPrefix' => '',
            'pConnect' => TRUE,
            'DBDebug'  => TRUE,
            'cacheOn'  => FALSE,
            'cacheDir' => '',
            'charset'  => 'utf8',
            'DBCollat' => 'utf8_general_ci',
            'swapPre'  => '',
            'encrypt'  => FALSE,
            'compress' => FALSE,
            'strictOn' => FALSE
        ]
];
```

28.2 Configurando através do arquivo .env

Para utilizar a configuração de banco de dados a partir de arquivo `.env` você deve criar esse arquivo no diretório raiz da aplicação e

adicionar a ele os mesmos parâmetros que foram passados no arquivo padrão.

O nome de cada parâmetro de configuração é definido usando 3 partes separadas por um ponto (.). A primeira parte corresponde à configuração à qual a linha corresponde, que nesse caso é `database` por se tratar do banco de dados. A segunda parte é o nome do grupo e a terceira é o parâmetro de configuração.

Veja a seguir um exemplo com alguns dos parâmetros de configuração:

```
database.default.username = 'jla_admin';
database.default.password = '1sW45@dsd';
database.default.database = 'livroci4';
```

28.3 Os parâmetros de configuração

Veja a seguir uma lista com mais informações sobre os parâmetros utilizados para configurar as conexões com bancos de dados.

- `dsn` : string de conexão com o banco de dados
- `hostname` : nome do host de conexão
- `username` : nome do usuário para autenticação
- `password` : senha do usuário para autenticação
- `database` : nome do banco de dados a ser conectado
- `DBDriver` : driver de conexão (por padrão vem definido como MySQLi)
- `DBPrefix` : prefixo opcional que será adicionado ao nome da tabela quando requisições utilizando o Query Builder forem executadas
- `pConnect` : determina se a conexão é persistente ou não (`true` ou `false`)
- `DBDebug` : determina se os erros relacionados ao banco de dados serão exibidos ou não (`true` ou `false`)

- `cacheOn` : ativa ou desativa o cache de consultas (`true` ou `false`)
- `cacheDir` : caminho absoluto do diretório onde o cache será armazenado
- `charset` : o charset utilizado na comunicação com o banco de dados
- `DBCollat` : conjunto de caracteres usado na comunicação com banco de dados (utilizado somente se o driver especificado for MySQLi)
- `swapPre` : um prefixo da tabela padrão que deve ser trocado com o `DBPrefix`
- `schema` : o esquema do banco de dados utilizado por drivers como PostgreSQL e ODBC (padrão: `public`)
- `encrypt` : se deve ou não utilizar uma conexão criptografada.
  - os drivers `sqlsrv` e `pdo/sqlsrv` aceitam `true` ou `false`
  - os drivers `MySQLi` e `pdo/mysql` aceitam uma matriz com as seguintes opções:
    - `ssl_key` : caminho para o arquivo de chave privada
    - `ssl_cert` : caminho para o arquivo de certificado de chave pública
    - `ssl_ca` : caminho para o arquivo de autoridade de certificação
    - `ssl_capath` : caminho para um diretório contendo certificados CA confiáveis no formato PEM
    - `ssl_cipher` : lista de cifras permitidas para serem usadas na criptografia, separadas por dois pontos (`:`)
    - `ssl_verify` : `true` ou `false` ; se deve verificar o certificado do servidor ou não (apenas `MySQLi`)
- `compress` : se deve ou não usar compressão (`true` ou `false` , somente para o driver `MySQL`)
- `strictOn` : determina se será utilizado o 'Strict Mode' para instruções SQL (`true` ou `false`)
- `port` : número da porta de conexão com o banco de dados (não vem definido no array, caso seja necessário utilizar, adicionar `$default['port'] = 5432;`)

ATENÇÃO

Dependendo da plataforma de banco de dados que a aplicação estiver utilizando, nem todos os valores serão necessários. Por exemplo, se estiver utilizando SQLite, não é necessário fornecer nome de usuário e senha, e o nome do banco de dados será o caminho para o arquivo de banco de dados.

Capítulo 29
Conectando a aplicação com o banco de dados

A conexão ao banco de dados no CodeIgniter pode ser realizada dentro de um método ou então a partir do método construtor da classe, ficando disponível de forma global para uso em todos os métodos que a compõem.

Para realizar essa conexão, basta inserir a linha a seguir dentro do método (`__constructor` ou não) da classe que deve se comunicar com o banco de dados.

```
$db = \Config\Database::connect();
```

No capítulo anterior você aprendeu como definir as configurações para acesso ao banco de dados e viu que é possível operar com grupos de conexões e que no arquivo de configuração podemos definir o grupo padrão.

Esse grupo padrão é utilizado sempre que o método `connect()` for chamado sem receber nenhum parâmetro. Então, se em algum momento dentro da sua aplicação você precisar se conectar com um outro grupo, basta passar o nome do grupo já configurado como parâmetro para `connect()` .

Uma alternativa a essa forma de conexão é o método `db_connect()` , que faz a mesma coisa que `\Config\Database::connect()` . O método `db_connect()` é apenas uma forma mais abreviada de realizar a conexão.

29.1 Múltiplas conexões para o mesmo banco de dados

Por padrão, o método `connect()` retorna sempre a mesma instância de conexão com o banco de dados, ou seja, se você precisar de uma conexão separada com o mesmo banco de dados, será necessário informar um segundo parâmetro para o método como `false`.

```
$db = \Config\Database::connect('grupo_de_configuracao', false);
```

Caso você precise apenas utilizar um banco de dados diferente na mesma conexão, basta utilizar o código a seguir para definir esse banco de dados:

```
$db->setDatabase('nome_banco_de_dados');
```

29.2 Conexão com configurações personalizadas

Você pode realizar conexões com bancos de dados utilizando configurações personalizadas que não tenham sido informadas nos grupos das configurações.

Para isso, basta que você crie um array no mesmo padrão do utilizado no arquivo de configuração padrão, preenchendo os valores com os da conexão que deseja realizar.

```
$personalizada = [
    'DSN'      => '',
    'hostname' => 'localhost',
    'username' => '',
    'password' => '',
    'database' => '',
    'DBDriver' => 'MySQLi',
    'DBPrefix' => '',
    'pConnect' => false,
    'DBDebug'  => (ENVIRONMENT !== 'production'),
    'cacheOn'  => false,
    'cacheDir' => '',
    'charset'  => 'utf8',
```

```
    'DBCollat' => 'utf8_general_ci',
    'swapPre'  => '',
    'encrypt'  => false,
    'compress' => false,
    'strictOn' => false,
    'failover' => [],
    'port'     => 3306,
];

$db = \Config\Database::connect($personalizada);
```

Capítulo 30
Operações básicas com banco de dados

Podemos considerar como operações básicas do trabalho com banco de dados as seguintes:

- SELECT
- INSERT
- UPDATE
- DELETE

No CodeIgniter, você pode executar essas operações básicas usando instruções SQL de forma textual (`SELECT * FROM autores`) ou utilizando os recursos do *Query Builder*, que será abordado no próximo capítulo.

Para executar uma consulta você pode utilizar o seguinte comando:

```
$query = $db->query('SELECT * FROM autores');
```

O método `query()` retorna um objeto com o resultado da consulta sempre que instruções SQL do tipo *leitura* (SELECT) são executadas. Se o tipo de instrução for *escrita* (INSERT, UPDATE ou DELETE) então o método `query()` retornará o status da operação (`true` ou `false`).

No exemplo acima foi executada uma instrução de leitura e os dados retornados armazenados na variável `$query` . Dessa mesma forma você pode realizar qualquer instrução SQL apenas passando como parâmetro para o método `query()` a instrução SQL correspondente aos dados que deseja obter ou escrever no banco de dados.

30.1 Estrutura e segurança

Trabalhando com prefixos de tabela

Caso você tenha configurado um prefixo para utilizar nas tabelas da sua aplicação, você pode utilizar o método `prefixTable()`, passando como parâmetro o nome da tabela, assim o prefixo será adicionado automaticamente à instrução SQL.

```
$db->prefixTable('autores');
```

Caso você precise alterar o prefixo de forma programática, sem criar uma nova conexão, deve utilizar o método `setPrefix()`, passando como parâmetro o prefixo a ser utilizado.

```
$db->setPrefix('pfx');
$db->prefixTable('autores');
```

Para saber qual prefixo está sendo utilizado, basta executar o método `getPrefix()`.

```
$DBPrefix = $db->getPrefix();
```

Protegendo tabelas e colunas do banco de dados

Para proteger os nomes de tabelas e as colunas do banco de dados no CodeIgniter você pode utilizar o método `protectIdentifiers()` passando como parâmetro o nome da tabela, sempre que estiver realizando operações sem o uso do *Query Builder*, que aplica essa proteção de forma automática.

```
$db->protectIdentifiers('autores');
```

O método `protectedIdentifiers()` também pode adicionar um prefixo previamente configurado à tabela, para isso você deve informar como segundo parâmetro o valor `true`.

Melhorando a segurança das operações com *escape*

Uma das boas práticas recomendadas é fazer o *escape* dos dados antes de enviá-los ao banco de dados, e com o CodeIgniter esse procedimento é facilitado, pois ele possui 3 métodos para nos auxiliar.

`$db->escape()`

Esse método faz o escape de dados para strings, adicionando aspas simples automaticamente à instrução.

```
$sql = "INSERT INTO autores (nome) VALUES(" . $db->escape($nome) . ")";
```

`$db->escapeString()`

Esse método faz o escape dos dados passados como parâmetro independente do tipo.

```
$sql = "INSERT INTO autores (nome) VALUES('" . $db->escapeString($nome) .
"')";
```

`$db->escapeLikeString()`

Esse método deve ser utilizado sempre que strings forem usadas junto de `LIKE`, para que os curingas (`%`, `_`) utilizados na string também sejam escapados de forma adequada.

```
$busca = 'José% Geraldo';
$sql = "SELECT id FROM autores WHERE nome LIKE '%" . $db-
>escapeLikeString($busca)."%' ESCAPE '!'";
```

> **ATENÇÃO**
>
> O método `escapeLikeString()` usa o ponto de exclamação (!) para escapar caracteres especiais para condições LIKE. Como o método escapa sequências parciais que seriam colocadas entre aspas, ele não pode adicionar de forma automática a condição `ESCAPE '!'`, sendo necessário fazer isso manualmente.

30.2 Instruções SQL utilizando bindings

O uso de *bindings* simplifica a sintaxe SQL das instruções, permitindo que você utilize os valores a serem transmitidos através de um array contendo os valores a serem substituídos na instrução SQL.

Para identificar na instrução os pontos que devem ser substituídos, você utilizará o ponto de interrogação (?). Cada ponto de interrogação corresponde a uma posição no array que é passado como parâmetro para o método `query()`.

```
$sql = "SELECT * FROM autores WHERE id = ? AND status = ? AND nome = ?";
$db->query($sql, [3, 1, 'Jonathan']);
```

Ao renderizar a substituição dos pontos de interrogação pelos valores do array, é respeitada a ordem em que a instrução foi construída, ou seja, o primeiro ponto de interrogação será substituído pelo primeiro elemento do array, e assim sucessivamente.

```
SELECT * FROM autores WHERE id = 3 AND status = 1 AND nome = 'Jonathan'
```

Também é possível utilizar um array dentro de uma posição do array principal para construir condições utilizando `IN`.

```
$sql = "SELECT * FROM autores WHERE id = ? AND status = ?";
$db->query($sql, [[3,5,7], 1);
```

O resultado para essa instrução seria:

```
SELECT * FROM autores WHERE id IN (3,5,7) AND status = 1
```

Um outro benefício do uso de *bindings* é que os valores contidos no array são escapados automaticamente no processo de construção e execução da instrução SQL.

Substituindo a '?' por nomes legíveis

Usar o ponto de interrogação é uma comodidade, mas dependendo da quantidade de informações a serem passadas pode ficar complicado entender o código no futuro. Pensando nisso, o CodeIgniter traz a implementação de *named bindings*.

Em vez de utilizar o ponto de interrogação, você passa a utilizar um nome definido como chave no array que contém os dados a serem substituídos na instrução. Esses nomes devem estar entre dois pontos (:).

```
$sql = "SELECT * FROM autores WHERE id = :id: AND status = :status: AND
nome = :nome:";
$db->query($sql, [
    'id'     => 3,
    'status' => 1,
    'nome'   => 'Jonathan'
]);
```

O resultado:

```
SELECT * FROM autores WHERE id = 3 AND status = 1 AND nome = 'Jonathan'
```

30.3 Manipulando resultados de instruções SQL

A manipulação de resultados de instruções SQL é fundamental para qualquer aplicação web, pois é através dessa manipulação que os dados a serem exibidos ao usuário serão formatados e enviados à view.

A seguir você verá os principais métodos para fazer a manipulação de resultados de forma eficiente e prática.

`getResult()`

Método responsável por retornar o resultado da instrução em um array de objetos ou, em caso de falha, um array vazio. É muito comum o uso desse método em um `foreach`.

```php
$query = $db->query('SELECT * FROM autores');
foreach($query->getResult() as $row){
    echo $row->id;
    echo $row->nome;
    echo $row->status;
}
```

Caso você prefira lidar com arrays ou sua aplicação necessite que os dados retornados sejam um array e não um objeto, basta passar o parâmetro `array` para o método `getResult()` e ele retornará os resultados como um array e não mais como um objeto.

```php
$query = $db->query('SELECT * FROM autores');
foreach($query->getResult('array') as $row){
    echo $row['id'];
    echo $row['nome'];
    echo $row['status'];
}
```

Existe uma outra forma de obter os resultados da instrução SQL em formato de array, que é utilizando o método `getResultArray()` em vez de passar o parâmetro `array` para o método `getResult()`.

Você também pode informar o nome que representa uma classe a ser instanciada passando como parâmetro o nome da classe.

```php
$query = $db->query('SELECT * FROM autores');
foreach($query->getResult('Autor') as $row){
    echo $autor->id;
    echo $autor->nome;
    echo $autor->status;
}
```

getRow()

Esse é o método responsável por retornar apenas uma linha do resultado da instrução SQL. Se for utilizado sem que um parâmetro seja passado, a primeira linha do resultado será retornada, mas se for passado um número como parâmetro, será retornada a linha correspondente a ele.

```php
$query = $db->query('SELECT * FROM autores');
$row = $query->getRow();

if(isset($row)){
    echo $row->id;
    echo $row->nome;
    echo $row->status;
}
```

O método `getRow()` aceita um segundo parâmetro que permite a informação de uma classe a ser instanciada para a respectiva linha retornada. Dessa forma você terá acesso tanto aos resultados da consulta quanto a métodos definidos na classe instanciada.

```php
$query = $db->query('SELECT * FROM autores');
$row = $query->getRow(0, 'Autor');

if(isset($row)){
    echo $row->id;
    echo $row->nome;
    echo $row->iniciais_autor; //método fictício que retornaria as iniciar
do nome do autor, por exemplo Jonathan Lamim retornaria JL
}
```

> **ATENÇÃO**
>
> Ao informar o número da linha a ser retornada é importante se lembrar que os arrays começam com o contador em zero, ou seja, se você quer o registro da primeira linha, precisa passar como parâmetro 0 (zero), se quer a segunda linha, informe 1 e assim por diante.

Se você preferir trabalhar com os resultados em formato array em vez de objeto, basta utilizar o método `getRowArray()` no lugar de `getRow()`. Também é possível utilizar algumas variações do método para obter linhas específicas do resultado da instrução SQL.

```php
$query = $db->query('SELECT * FROM autores');

$row = $query->getFirstRow(); // retorna a primeira linha
$row = $query->getLastRow(); // retorna a última linha
$row = $query->getNextRow(); // retorna a próxima linha
$row = $query->getPreviousRow(); // retorna a linha anterior
```

E para esses métodos você pode informar o parâmetro `array` caso queira o resultado em formato de array.

Os métodos apresentados carregam o resultado inteiro na memória, como se fosse uma pré-busca. Então se você for processar um grande conjunto de resultados, utilize o método `getUnbufferedRow()` .

`getUnbufferedRow()`

Esse método retorna uma linha única de resultado sem utilizar as informações da pré-busca que estiverem na memória, como acontece no método `row()` . Caso sua consulta tenha mais de uma linha, ele vai retornar a linha atual e avançar com o ponteiro de dados interno.

```php
$query = $db->query('SELECT * FROM autores');

while($row = $query->getUnbufferedRow()){
    echo $row->id;
    echo $row->nome;
    echo $row->status;
}
```

Por padrão, os dados retornados serão no formato de objeto, mas você pode passar como parâmetro a string `array` , que ele retornará os dados como um array.

30.4 Customizando resultados no formato de objeto

Como já vimos neste capítulo, é possível obter resultados de instruções SQL em uma instância de classe personalizada, ao invés de um array ou objeto.

Se a classe que foi passada como parâmetro não estiver carregada na memória, o `Autoloader` se encarregará disso automaticamente.

O objeto de classe que será construído conterá todos os valores retornados pela instrução SQL definidos como propriedades da classe. Se esses valores forem declarados e não forem públicos, será necessário um método `__set()` na classe para que os valores sejam definidos.

```
class Autor
{
    public $id;
    public $nome;
    public $status;
    protected $ultimo_acesso;

    public function ultimoAcesso($format)
    {
        return $this->ultimoAcesso->format($format);
    }

    public function __set($name, $value)
    {
        if ($name === 'ultimoAcesso')
        {
            $this->ultimoAcesso = DateTime::createFromFormat('U', $value);
        }
    }

    public function __get($name)
    {
        if (isset($this->$name))
        {
            return $this->$name;
        }
```

```
    }
}
```

getCustomRowObject()

O método retorna os resultados de uma instrução SQL como um array de instâncias da classe, e o único parâmetro necessário é o nome da classe.

```
$query = $db->query('SELECT * FROM autores');
$rows = $query->getCustomRowObject('Autor');

foreach($rows as $row){
    echo $autor->id;
    echo $autor->nome;
    echo $autor->ultimoAcesso('Y-m-d');
}
```

getCustomRowObject()

O método retorna uma única linha do conjunto de resultados de uma instrução SQL. O primeiro parâmetro é a linha desejada e o segundo, o nome da classe.

```
$query = $db->query('SELECT * FROM autores');
$row = $query->getCustomRowObject('Autor');

if(isset($row)){
    echo $autor->nome;
    echo $autor->ultimoAcesso('Y-m-d');
}
```

30.5 Métodos auxiliares

Veja a seguir alguns métodos auxiliares que você pode utilizar em suas aplicações para agilizar o processo de desenvolvimento.

getFieldCount()

Retorna a quantidade de campos retornados em uma consulta.

```
$query = $db->query('SELECT * FROM autores');
echo $query->getFieldCount();
```

getFieldNames()

Retorna os nomes dos campos de uma consulta organizados em um array.

```
$query = $db->query('SELECT * FROM autores');
echo $query->getFieldNames();
```

freeResult()

Esse método libera a memória associada ao resultado e exclui o ID do recurso do resultado. Ele é muito útil quando você está executando muitas consultas em um script específico, pois se chamado após a geração de cada resultado, ele libera a memória, reduzindo seu consumo e gerando melhoria de performance para a aplicação.

```
$query = $this->db->query('SELECT * FROM autores');

foreach ($query->getResult() as $row)
{
    echo $row->nome;
}

$query->freeResult();

$query2 = $db->query('SELECT titulo FROM posts');

$row = $query2->getRow();
echo $row->titulo;

$query2->freeResult();
```

A seguir você verá 3 métodos bastante úteis quando estiver executando suas instruções SQL.

insertID()

Esse método retorna o ID do registro inserido no banco de dados através do método `insert()`.

```
$builder = $this->db->table('autores');

$autor = array(
    'nome'   => 'José Antônio',
    'status' => 1
);

$builder->insert('autores', $autor);
echo $builder->insertID();
```

> **ATENÇÃO**
>
> Se você estiver utilizando PDO com o PostgreSQL ou mesmo o Interbase, este método necessitará de um parâmetro `$name`, que vai especificar a sequência apropriada para a verificação do ID inserido.

`affectedRows()`

Esse método exibe o número de linhas que foram afetadas ao executar uma instrução SQL de gravação (INSERT, UPDATE, DELETE).

```
$builder = $this->db->table('autores');

$autor = array(
    'nome'   => 'José Antônio',
    'status' => 1
);

$builder->insert('autores', $autor);
echo $builder->affectedRows();
```

`getLastQuery()`

Esse método exibe a string da última instrução SQL executada.

```php
$builder = $this->db->table('autores');

$autor = array(
    'nome'   => 'José Antônio',
    'status' => 1
);

$builder->insert('autores', $autor);
echo $builder->getLastQuery();
```

countAll()

Esse método retorna a quantidade de registros em uma tabela, que deve ser informada como parâmetro.

```php
echo $this->db->countAll('autores');
```

Também é possível executar da seguinte forma:

```php
$builder = $this->db->table('autores');
echo $builder->conuntAll();
```

Capítulo 31
Query Builder

A classe Query Builder no CodeIgniter permite que você execute operações com bancos de dados de forma rápida e com o mínimo de scripts. Em vez de exigir que cada tabela do banco de dados seja seu próprio arquivo de classe, o CodeIgniter fornece uma interface mais simplificada e com instruções SQL mais seguras.

O carregamento do Query Builder é feito através do método `table()` na conexão com o banco de dados, uma vez que esse método define o bloco `FROM` da instrução SQL, retornando uma nova instância da classe Query Builder. Como a classe é carregada na memória apenas quando você solicita de forma específica, nenhum recurso é utilizado por padrão antes da sua solicitação.

Veja a seguir um exemplo de como instanciar a classe Query Builder em sua aplicação.

```
$db = \Config\Database::connect();
$builder = $db->table('autores');
```

Ao criar os models da sua aplicação, você pode instanciar a classe Query Builder no método `__construct()`, assim você não precisará repetir as linhas de código acima sempre que for necessário executar uma instrução SQL através do Query Builder.

```
<?php namespace App\Models;

    use CodeIgniter\Database\ConnectionInterface;

    class AutorModel
    {
        protected $db;
        private $builder;

        public function __construct(ConnectionInterface &$db)
```

```
        {
            $this->db =& $db;
            $this->builder = $this->db->table('autores');
        }
    }
```

> **ATENÇÃO**
>
> Para os exemplos de código deste capítulo vamos considerar que a classe do Query Builder foi instanciada no método `__construct()`, portanto chamaremos diretamente `$this->builder`.

31.1 Métodos de seleção

A seguir você verá um conjunto de métodos que podem ser utilizados nas instruções SQL para seleção de informações no banco de dados.

`get()`

O método `get()` executará a instrução SQL retornando os resultados.

```
$query = $this->builder->get();
// SQL: SELECT * FROM autores
```

Você também pode informar 2 parâmetros para o método `get()` para limitar a quantidade de registros a serem retornados. Esses parâmetros são muito úteis quando há a necessidade de fazer paginação dos registros na aplicação.

O primeiro parâmetro é a quantidade máxima de registros (LIMIT) e o segundo é a "página" (OFFSET).

```
$query = $this->builder->get(10, 5);
// SQL: SELECT * FROM autores LIMIT 5, 10
```

Para exibir os resultados na sua aplicação você utilizará o método `getResult()`, que foi visto no capítulo anterior.

```
$query = $this->builder->get(10, 5);

foreach($query->getResult() as $row)
{
    echo $row->nome;
}
```

`getWhere()`

Método muito parecido com o `get()` exceto pelo fato de permitir a definição de parâmetros para a consulta. Aceita 3 parâmetros: o primeiro é o conjunto de parâmetros da consulta definidos em um array, o segundo é a quantidade máxima de registros (LIMIT) e o terceiro, a "página" (OFFSET).

```
$parametros = ['status' => 1];
$query = $this->builder->getWhere($parametros, 10, 5);
// SQL: SELECT * FROM autores WHERE status = 1 LIMIT 5, 10
```

`select()`

Método que permite definir quais colunas da tabela serão retornadas na consulta. Ele não executa a instrução, apenas adiciona um conjunto de informações a ela, sendo necessário o uso do método `get()` para o retorno dos dados.

```
$this->builder->select('id, nome, status');
$query = $this->builder->get();
// SQL: SELECT id, nome, status FROM autores
```

Só é necessário o uso do método `select()` quando você quiser retornar apenas algumas colunas da tabela, pois por padrão o Query Builder já constrói a instrução SQL para retornar todas as colunas da tabela.

ATENÇÃO

O CodeIgniter faz automaticamente o escape das instruções para manter o banco de dados em segurança e pode ser que em algum momento seja necessário executar alguma instrução sem essa proteção para alcançar os resultados desejados, para isso você pode usar um segundo parâmetro disponível para o método `select()`, que pode ser definido como `FALSE` caso queira desativar a proteção para a instrução a ser executada.

selectMax()

Assim como o método `select()`, este método permite definir uma parte da instrução SQL que no caso é o campo com maior valor dentre os registros selecionados. Ele aceita 2 parâmetros, o primeiro é a coluna a ser analisada e o segundo (opcional) é o nome personalizado da coluna que exibirá a instrução.

```
$this->builder->selectMax('idade');
$query = $this->builder->get();
// SQL: SELECT MAX(idade) as idade FROM autores

$this->builder->selectMax('idade', 'idade_autor');
$query = $this->builder->get();
// SQL: SELECT MAX(idade) as idade_autor FROM autores
```

selectMin()

Funciona da mesma forma que `selectMax()`, com a mesma estrutura de parâmetros, porém selecionando o registro de menor valor dentro de todos os registros retornados.

```
$this->builder->selectMin('idade');
$query = $this->builde->get();
// SQL: SELECT MIN(idade) as idade FROM autores
```

selectAvg()

Funciona da mesma maneira que os métodos `selectMax()` e `selectMin()`, porém, ele calcula a média aritmética dos valores em uma única coluna.

```
$this->builder->selecAvg('idade');
$query = $this->builde->get();
// SQL: SELECT AVG(idade) as idade FROM autores
```

selectSum()

Ainda seguindo a linha de `selectMax()`, `selectMin()` e `selectAvg()`, este soma os valores dos registros para a coluna especificada e armazena em uma nova coluna.

```
$this->builder->selecSum('idade');
$query = $this->builde->get();
// SQL: SELECT SUM(idade) as idade FROM autores
```

selectCount()

Idem aos 4 anteriores, ele conta a quantidade de registros selecionados pela instrução.

```
$this->builder->selecCount('idade');
$query = $this->builde->get();
// SQL: SELECT COUNT(idade) as idade FROM autores
```

from()

Esse método permite informar a origem (tabela) onde os dados serão buscados. Caso você já tenha informado a tabela no método `__construct()` através de `$this->db->table()`, não é necessário utilizar o método `from()` para buscar dados na mesma tabela. Se você precisar utilizar outras tabelas além da já informada, basta chamar o método passando o nome da tabela como parâmetro.

```
$this->builder->select('id, nome, status');
$this->builder->from('autores');
$query = $this->builder->get();
// SQL: SELECT id, nome, status FROM autores
```

join()

Através desse método você constrói consultas com o uso da instrução `JOIN`. Se na sua instrução você precisar de mais de um `JOIN`, basta repetir a chamada ao método com os parâmetros necessários para sua consulta.

Se a sua consulta precisar utilizar um tipo específico de `JOIN` (`left`, `right`, `outer`, `inner`, `left outer` ou `right outer`), basta informar no terceiro parâmetro qual o tipo desejado, enquanto no primeiro, você informa a tabela e no segundo, as instruções para consulta através do `JOIN`.

```
$this->builder->select('*');
$this->builder->join('posts', 'posts.id_autor = autores.id');
$query = $this->builder->get();
// SQL: SELECT * FROM autores JOIN posts ON posts.id_autor = autores.id

$this->builder->select('*');
$this->builder->join('posts', 'posts.id_autor = autores.id', 'left');
$query = $this->builder->get();
// SQL: SELECT * FROM autores LEFT JOIN posts ON posts.id_autor =
autores.id
```

Métodos de seleção para dados específicos

`where()`

O método mais tradicional para adicionar filtros em uma instrução SQL. Ele recebe dois parâmetros: o primeiro é o nome do campo e o segundo, o valor a ser comparado. E se sua aplicação precisar de múltiplos filtros, basta repetir a chamada ao método `where()`.

```
$this->builder->where('status', 1);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1

$this->builder->where('status', 1);
$this->builder->where('id', 3);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 AND id = 3
```

Note que no exemplo não foi informada a forma de comparação entre o parâmetro e o valor do campo, nesse caso, o CodeIgniter interpreta como sendo uma comparação para valores iguais. Mas você pode customizar a comparação utilizando o sinal comparador junto do primeiro parâmetro.

```
$this->builder->where('status', 1);
$this->builder->where('id >', 3);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 AND id > 3
```

Existem situações dentro das aplicações onde a quantidade de filtros é relativamente grande e ficar repetindo `$this->builder->where()` deixa o código bastante extenso. Pensando nisso, o CodeIgniter permite que você otimize a escrita de código utilizando um array associativo para definir todas as cláusulas `where` e passando esse array como parâmetro para o método `where()`.

```
$parametros = [
    'status' => 1,
    'id >'   => 3
];
$this->builder->where($parametros);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 AND id > 3
```

Além de todas essas possibilidades com o método `where()` você ainda tem outra utilidade para o caso de preferir usar uma `string` com a instrução para a cláusula `where`, em vez dos parâmetros permitidos pelo método.

```
$instrucao = "status = 1 AND id > 3";
$this->builder->where($instrucao);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 AND id > 3
```

Outra situação bastante vista em aplicações é a necessidade de executar uma consulta dentro da outra, recurso também chamado de *subquery*. No CodeIgniter 4, esse processo ficou mais simples

pois agora é possível utilizar uma função anônima para criar essa *subquery*.

A estrutura de uso do método `where()` não muda muito. O primeiro parâmetro continua sendo o campo a ser comparado enquanto o segundo recebe a função anônima que retornará os dados da subquery usando os métodos do Query Builder, da mesma forma como vimos até o momento.

```
$this->builder->where('id <', function(BaseBuilder $builder) {
    return $builder->select('MAX(id_autor)', false)->from('posts')-
>where('status', 1);
});
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE id < (SELECT MAX(id_autor) FROM posts
WHERE status = 1)
```

> **ATENÇÃO**
>
> Assim como no método `select()`, aqui também pode ser necessário executar alguma instrução sem a proteção que o CodeIgniter aplica automaticamente, para isso você pode usar um terceiro parâmetro disponível para o método `where()`, que pode ser definido como `FALSE` caso queira desativar a proteção para a instrução a ser executada.

`orWhere()`

Esse método tem sua composição e parâmetros idênticos ao método `where()`, porém a instrução SQL originada com o uso desse método vai utilizar `OR` em vez de `AND`. Esse método é bastante útil quando os resultados a serem obtidos não precisarem ser completamente compatíveis com todos os filtros aplicados.

```
$this->builder->where('status', 1);
$this->builder->orWhere('id >', 5);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 OR id > 3
```

whereIn()

O método `whereIn()` tem sua estrutura muito similar à dos métodos `where()` e `orWhere()`, porém ele faz uma comparação com um conjunto de dados que é passado através de um array.

```
$autores = [1,2,5,7,8];
$this->builder->whereIn('id', $autores);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE id IN (1, 2, 5, 7, 8)
```

Esse método também permite o uso de função anônima para a execução de subquery, sendo estruturado da mesma maneira que no método `where()`.

```
$this->builder->whereIn('id', function(BaseBuilder $builder) {
    return $builder->select('id_autor')->from('posts')->where('status',
1)->groupBy('id_autor');
});
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE id IN (SELECT id_autor FROM posts
WHERE status = 1 GROUP BY id_autor)
```

orWhereIn()

Esse método funciona como uma combinação entre o método `orWhere()` e o `whereIn()`, aceitando a mesma estrutura de parâmetros e até mesmo a função anônima para uso de subquery.

```
$autores = [1,2,5,7,8];
$this->builder->where('status', 1);
$this->builder->orWhereIn('id', $autores);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 OR id IN (1, 2, 5, 7, 8)
```

whereNotIn()

É como o `whereIn()`, porém ao contrário, ou seja, retorna todos os resultados diferentes dos informados no filtro. Ele também aceita

função anônima para uso de subquery.

```
$autores = [1,2,5,7,8];
$this->builder->whereNotIn('id', $autores);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE id NOT IN (1, 2, 5, 7, 8)
```

orWhereNotIn()

Uma combinação de `orWhere()` com `whereNotIn()`, aceitando a mesma estrutura de parâmetros e a função anônima para uso de subquery.

```
$autores = [1,2,5,7,8];
$this->builder->where('status', 1);
$this->builder->orWhereNotIn('id', $autores);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE status = 1 OR id NOT IN (1, 2, 5, 7,
8)
```

Métodos de seleção para dados similares

like()

Esse método adiciona a cláusula `LIKE` à instrução SQL e, por segurança, todos os valores passados como parâmetro são escapados. O primeiro parâmetro se refere à coluna no banco de dados, o segundo, ao conteúdo a ser pesquisado.

```
$this->builder->like('nome', 'Jonathan');
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE nome LIKE '%Jonathan%' ESCAPE '!'
```

Repare que o wildcard (`%`) foi inserido de forma automática antes e depois do conteúdo a ser pesquisado. Pode ser que você precise utilizar o wildcard de outra forma para atender às necessidades da sua aplicação, e para isso basta utilizar o terceiro parâmetro do método `like()` para informar a posição do wildcard (`before` , `after` ou `both`).

```php
$this->builder->like('nome', 'Jonathan', 'before');
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE nome LIKE '%Jonathan' ESCAPE '!'
```

Também é possível utilizar um array associativo para informar os parâmetros ao método.

```php
$parametros = [
    'nome' => 'Jonathan',
    'bio'  => 'Escritor'
];
$this->builder->like($parametros);
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE nome LIKE '%Jonathan%' ESCAPE '!' AND
bio LIKE '%Escritor%' ESCAPE '!'
```

orLike()

Método idêntico ao `like()`, exceto pelo fato de adicionar múltiplas cláusulas `LIKE` separadas por `OR`.

```php
$this->builder->like('nome', 'Jonathan');
$this->builder->orLike('bio', 'Escritor');
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE nome LIKE '%Jonathan%' ESCAPE '!' OR
bio LIKE '%Escritor%' ESCAPE '!'
```

notLike()

Método idêntico ao `like()`, exceto pelo fato de adicionar um `NOT LIKE` à instrução SQL.

```php
$this->builder->notLike('nome', 'Jonathan');
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE nome NOT LIKE '%Jonathan%' ESCAPE '!'
```

orNotLike()

Método idêntico ao `notLike()`, exceto pelo fato de adicionar um `OR` à instrução SQL.

```php
$this->builder->like('nome', 'Jonathan');
$this->builder->orNotLike('bio', 'Escritor');
```

```
$query = $this->builder->get();
// SQL: SELECT * FROM autores WHERE nome LIKE '%Jonathan%' ESCAPE '!' OR
bio NOT LIKE '%Escritor%' ESCAPE '!'
```

groupBy()

Esse método permite agrupar os resultados a partir de uma coluna da tabela onde a pesquisa está sendo realizada.

```
$this->builder->groupBy('uf');
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf
```

É possível passar múltiplas colunas para agrupamento através de um array.

```
$agrupar = ['uf', 'cidade'];
$this->builder->groupBy($agrupar);
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf, cidade
```

distinct()

Esse método adiciona a palavra-chave `DISTINCT` à instrução SQL.

```
$this->builder->distinct();
$query = $this->builder->get();
// SQL: SELECT DISTINCT * FROM autores
```

having()

Esse método permite adicionar a cláusula `HAVING` à instrução SQL e pode ser escrito de 2 formas, além da possibilidade de uso com array associativo.

```
$this->builder->groupBy('uf');
$this->builder->having('uf','ES');
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf HAVING uf = 'ES'
```

Além da sintaxe `$this->builder->having('uf','ES');` você também pode utilizar `$this->builder->having('uf = ES');` ou então com o array associativo.

```
$agrupar = ['uf', 'cidade'];
$this->builder->groupBy($agrupar);
$having = [
    'uf'         => 'ES',
    'cidade !=' => 'Vila Velha'
];
$this->builder->having($having);
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf, cidade HAVING uf = 'ES', cidade
!= 'Vila Velha'
```

> **ATENÇÃO**
>
> Assim como nos métodos `select()` e `where()`, aqui também pode ser necessário executar alguma instrução sem a proteção que o CodeIgniter aplica automaticamente. Para isso você pode usar um terceiro parâmetro disponível para o método `having()`, que pode ser definido como `FALSE` caso queira desativar a proteção para a instrução a ser executada.

orHaving()

Método idêntico ao `having()`, exceto pelo fato de adicionar um `OR` à instrução SQL.

```
$agrupar = ['uf', 'cidade'];
$this->builder->groupBy($agrupar);
$this->builder->having('uf','ES');
$this->builder->orHaving('cidade','Vila Velha');
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf HAVING uf = 'ES' OR cidade =
'Vila Velha'
```

havingIn()

Esse método aplica a cláusula `HAVING` na instrução SQL para buscar por um conjunto específico de informações que são passadas como um array no segundo parâmetro.

```
$having = ['ES', 'RJ', 'SP'];
$this->builder->groupBy('uf');
$this->builder->havingIn('uf', $having);
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf HAVING uf IN ('ES', 'RJ', 'SP')
```

O método `havingIn()` também permite o uso de função anônima para a execução de subquery da mesma forma como já vimos em outros métodos do Query Builder.

```
$this->builder->groupBy('uf');
$this->builder->havingIn('uf', function(BaseBuilder $builder) {
    return $builder->select('nome')->from('estados')->where('status', 1);
});
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf HAVING uf IN (SELECT nome FROM
estados WHERE status = 1)
```

orHavingIn()

Esse método é uma combinação dos métodos `orHaving()` e `havingIn()`, e também aceita o uso de função anônima.

```
$agrupar = ['cidade', 'uf'];
$having = ['ES', 'RJ', 'SP'];
$this->builder->groupBy($agrupar);
$this->builder->having('cidade','Vila Velha');
$this->builder->orHavingIn('uf', $having);
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY cidade, uf HAVING cidade = 'Vila
Velha' OR uf IN ('ES', 'RJ', 'SP')
```

havingNotIn()

Método idêntico ao `havingIn()`, exceto pelo fato de adicionar um `NOT` à instrução SQL. Também aceita função anônima.

```
$having = ['ES', 'RJ', 'SP'];
$this->builder->groupBy('uf');
```

```php
$this->builder->havingNotIn('uf', $having);
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY uf HAVING uf NOT IN ('ES', 'RJ',
'SP')
```

orHavingNotIn()

Uma combinação dos métodos `orHaving()` e `havingNoteIn()` que também aceita função anônima.

```php
$agrupar = ['cidade', 'uf'];
$having = ['ES', 'RJ', 'SP'];
$this->builder->groupBy($agrupar);
$this->builder->having('cidade','Vila Velha');
$this->builder->orHavingNotIn('uf', $having);
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY cidade, uf HAVING cidade = 'Vila
Velha' OR uf NOT IN ('ES', 'RJ', 'SP')
```

havingLike()

Esse método adiciona a cláusula `LIKE` junto da estrutura da cláusula `HAVING` dando mais flexibilidade para a obtenção dos resultados.

A forma de uso é idêntica à do método `like()`, permitindo informar a posição do wildcard (`before`, `after`, `both`) e utilizar um array associativo.

```php
$this->builder->groupBy('cidade');
$this->builder->havingLike('cidade','Rio');
$query = $this->builder->get();
// SQL: SELECT * FROM autores GROUP BY cidade HAVING cidade LIKE '%Rio%'
ESCAPE '!'
```

O método `havingLike()` possui outras 3 variações: `orHavingLike()`, `notHavingLike()` e `orNotHavingLike()`. Essas variações funcionam da mesma forma que os métodos `orHaving()`, `notHaving()` e `orNotHaving()` respectivamente, a única diferença é que será adicionada a estrutura da cláusula `LIKE` à instrução SQL.

ATENÇÃO

A cláusula `HAVING` é diferente da cláusula `WHERE`. Enquanto `WHERE` restringe os resultados da pesquisa após o uso de `FROM`, `HAVING` filtra o retorno de um grupo de registros. Sendo assim, o método `having()` e seus derivados devem ser sempre utilizados em conjunto com o método `groupBy()`.

31.2 Ordenação, limitação e contagem de resultados

`orderBy()`

Esse método permite a ordenação dos resultados através das consultas realizadas com métodos do Query Builder. E se for necessário alterar a forma como os dados são ordenados no resultado, basta informar `ASC` (ascendente), `DESC` (descendente) ou `RANDOM` (randômico) como segundo parâmetro do método, já que o primeiro é a coluna que servirá de base para a ordenação.

```
$this->builder->orderBy('nome', 'ASC');
$query = $this->builder->get();
// SQL: SELECT * FROM autores ORDER BY nome ASC
```

Você também pode informar múltiplas colunas para agrupamento com o respectivo parâmetro de ordenação usando uma única chamada ao método `orderBy()` ou com múltiplas chamadas.

```
$this->builder->orderBy('id DESC, nome ASC');
$query = $this->builder->get();
// SQL: SELECT * FROM autores ORDER BY id DESC, nome ASC

$this->builder->orderBy('id', 'DESC');
$this->builder->orderBy('nome', 'ASC');
```

```
$query = $this->builder->get();
// SQL: SELECT * FROM autores ORDER BY id DESC, nome ASC
```

limit()

Esse método permite limitar a quantidade de registros e é muito útil na paginação de resultados. Algo semelhante já foi visto neste capítulo no método `get()` .

O primeiro parâmetro é a quantidade máxima de registros (LIMIT) e o segundo parâmetro é a "página" (OFFSET).

```
$this->builder->limit(10, 5);
$query = $this->builder->get();
// SQL: SELECT * FROM autores LIMIT 5, 10
```

countAllResults()

Esse método permite a identificação do número de resultados obtidos em uma consulta através do Query Builder. Ele sempre retornará como resultado um número inteiro maior ou igual a zero. Você também pode utilizar o método `contuAllResults()` junto dos demais métodos do Query Builder.

```
$this->builder->where('status', 1);
$this->builder->countAllResults();
```

countAll()

Esse método permite a identificação do número total de registros em uma tabela do banco de dados.

```
$this->builder->countAll();
```

31.3 Agrupamento de consultas

O agrupamento de consultas permite a você criar grupos de cláusulas `WHERE` colocando-as entre parênteses e assim construir consultas mais complexas. Também é possível utilizar grupos

aninhados, o que aumenta ainda mais as possibilidades na construção de instruções SQL.

```
$this->builder->select('*')
              ->groupStart()
                  ->where('uf', 'ES')
                  ->orGroupStart()
                      ->where('cidade', 'Vila Velha')
                      ->where('cidade !=', 'Cariacica')
                  ->groupEnd()
              ->groupEnd()
              ->where('status', 1)
              ->get();
// SQL: SELECT * FROM autores WHERE ( uf = 'ES' OR ( cidade = 'Vila Velha'
AND cidade != 'Cariacica' ) ) AND status = 1
```

Algumas cláusulas SQL possuem métodos de abertura e fechamento específicos, mas a forma de estruturação continua a mesma do exemplo anterior.

`groupStart()` : utilizado para abrir os parênteses para o uso do método `where()` e seus derivados; `orGroupStart()` : utilizado para abrir os parênteses para o uso do método `where()` e seus derivados adicionando o prefixo `OR` ; `notGroupStart()` : utilizado para abrir os parênteses para o uso do método `where()` e seus derivados adicionando o prefixo `NOT` ; `orNotGroupStart()` : utilizado para abrir os parênteses para o uso do método `where()` e seus derivados adicionando o prefixo `OR NOT` ; `groupEnd()` : utilizado para fechar os parênteses para o uso do método `where()` e seus derivados; `groupHavingStart()` : utilizado para abrir os parênteses para o uso do método `having()` e seus derivados; `orGroupHavingStart()` : utilizado para abrir os parênteses para o uso do método `having()` e seus derivados adicionando o prefixo `OR` ; `notGroupHavingStart()` : utilizado para abrir os parênteses para o uso do método `having()` e seus derivados adicionando o prefixo `NOT` ; `orNotGroupHavingStart()` : utilizado para abrir os parênteses para o uso do método `having()` e seus derivados adicionando o prefixo `OR NOT` ; `groupHavingEnd()` :

utilizado para fechar os parênteses para o uso do método `having()` e seus derivados.

> **ATENÇÃO**
>
> Sempre que utilizar o agrupamento de consultas certifique-se de que cada grupo está entre os métodos `groupStart()` e `groupEnd()` ou seus derivados, caso contrário, a consulta não funcionará e a aplicação retornará um erro.

Agora que você já conhece os métodos do Query Builder para construir suas instruções SQL para pesquisa de dados, tem uma dica muito útil que precisa saber: **não é necessário repetir `$this->builder` para cada método que vai compor a instrução SQL**. É possível trabalhar com encadeamento de métodos no Query Builder do CodeIgniter 4.

Veja no exemplo a seguir como você deve trabalhar com o encadeamento de métodos:

```
$query = $this->builder->select('nome')
                       ->where('status', 1)
                       ->limit(10, 5)
                       ->get();
// SQL: SELECT nome FROM autores WHERE status = 1 LIMIT 5, 10
```

Outra coisa que você precisa saber é que existe a possibilidade de aplicar um *reset* no Query Builder, e isso é muito útil quando você está utilizando o Query Builder para gerar strings SQL através do método `getCompiledSelect()` e em algum ponto dentro do código decide executar a consulta.

Veja no exemplo a seguir que inicialmente é executado o método `getCompiledSelect()` com outros métodos do Query Builder para montar uma instrução SQL sem executar de fato essa instrução. Logo depois, é executada uma chamada ao método `getResultArray()` onde a intenção é retornar todos os registros da tabela `autores` no

formato de array. Porém, a instrução SQL executada para retornar os dados manteve as informações passadas para os métodos `select()` e `where()` encadeados com o método `getCompiledSelect()`, afetando o resultado da operação.

```
$sql = $this->builder->select('id, nome')
                     ->where('status', 1)
                     ->getCompiledSelect();
// SQL retornado por getCompiledSelect: SELECT id, nome FROM autores WHERE
status = 1

$dados = $this->builder->get()->getResultArray();
// SQL da instrução executada para obter os dados: SELECT id, nome FROM
autores WHERE status = 1
```

Para resolver esse problema e poder executar ambos os métodos sem que `getResultArray()` acabe herdando os parâmetros da consulta que foram utilizados junto com `getCompiledSelect()`, você pode utilizar o método `resetQuery()` entre as execuções ou então passar o parâmetro `FALSE` para o método `getCompiledSelect()`, assim ele será executado e automaticamente chamará o método `resetQuery()`.

```
$sql = $this->builder->select('id, nome')
                     ->where('status', 1)
                     ->getCompiledSelect(false);
// SQL retornado por getCompiledSelect: SELECT id, nome FROM autores WHERE
status = 1

$dados = $this->builder->get()->getResultArray();
// SQL da instrução executada para obter os dados: SELECT * FROM autores

$sql = $this->builder->select('id, nome')
                     ->where('status', 1)
                     ->getCompiledSelect();

$this->builder->resetQuery();

$dados = $this->builder->get()->getResultArray();
```

```
// SQL da instrução executada para obter os dados: SELECT id, nome FROM
autores WHERE status = 1
```

31.4 Inserção, atualização e remoção de dados

`set()`

Esse é um método auxiliar que permite a você informar os dados que devem ser inseridos ou atualizados no banco de dados através dos métodos `insert()` e `update()`.

```
$this->builder->set('nome','Pedro Antônio');
$this->builder->insert();
// SQL: INSERT INTO autores (nome) VALUES ('Pedro Antônio')
```

Além de o método poder ser chamado quantas vezes forem necessárias para adicionar os dados à instrução, também é possível passar os dados como um array multidimensional ou objeto e ainda definir se os dados devem ou não ser escapados, como já vimos neste capítulo. Para isso, basta informar o terceiro parâmetro como `FALSE`.

```
// MÚLTIPLOS `set()`
$this->builder->set('nome','Pedro Antônio');
$this->builder->set('status',1);
$this->builder->insert();
// SQL: INSERT INTO autores (nome) VALUES ('Pedro Antônio')

// ARRAY MULTIDIMENSIONAL
$autor = [
    'nome'   => 'Alberto Júnior',
    'cidade' => 'Colatina',
    'uf'     => 'ES',
    'bio'    => 'Analista de Sistemas'
    'status' => 1
];
$this->builder->set($autor);
$this->builder->insert();
```

```
// SQL: INSERT INTO autores (nome, cidade, uf, bio, status) VALUES
('Alberto Júnior', 'Colatina', 'ES', 'Analista de Sistemas', 1)

// DESATIVANDO O ESCAPE
$this->builder->set('nome','Pedro Antônio', FALSE);
$this->builder->set('status',1);
$this->builder->insert();
// SQL: INSERT INTO autores (nome) VALUES (Pedro Antônio)
```

insert()

Esse método faz a inserção de conteúdo no banco de dados a partir de um array multidimensional ou objeto passado como parâmetro.

```
$autor = [
    'nome'   => 'Carlos Alberto',
    'cidade' => 'Vila Velha',
    'uf'     => 'ES',
    'bio'    => 'Desenvolvedor frontend'
    'status' => 1
];
$this->builder->insert($autor);
// SQL: INSERT INTO autores (nome, cidade, uf, bio, status) VALUES
('Carlos Alberto', 'Vila Velha', 'ES', 'Desenvolvedor frontend', 1)
```

Repare que diferente da maioria do outros métodos do Query Builder vistos até o momento, esse não precisa de uma chamada ao método `get()` para gerar o resultado.

Tanto o `insert()` quanto os demais métodos que serão vistos daqui para a frente executam a instrução SQL logo que são chamados.

getCompiledInsert()

Parecido com o método `insert()`, porém ele não executa a instrução SQL inserindo o conteúdo no banco de dados, apenas retorna a string correspondente à instrução SQL.

```
$autor = [
    'nome'   => 'Carlos Alberto',
    'cidade' => 'Vila Velha',
    'uf'     => 'ES',
```

```
    'bio'    => 'Desenvolvedor frontend'
    'status' => 1
];
$sql = $this->builder->set($autor)->getCompiledInsert();
// SQL: INSERT INTO autores (nome, cidade, uf, bio, status) VALUES
('Carlos Alberto', 'Vila Velha', 'ES', 'Desenvolvedor frontend', 1)
```

O método `getCompiledSelect()` aceita como primeiro parâmetro o nome da tabela no banco de dados onde o conteúdo deve ser salvo e como segundo parâmetro a informação sobre se deve (`true`) ou não (`false`) executar de forma automática o método `resetQuery()` logo após sua execução. Por padrão, sempre que esse método é chamado, ele executa o método `resetQuery()` ao final da execução.

Veja no exemplo a seguir 2 chamadas seguidas ao método `getCompiledInsert()` , cada uma passando dados diferentes em `set()` . Na primeira chamada, é informado o segundo parâmetro para o método de modo que ele não execute o `resetQuery()` . E na segunda chamada, nenhum parâmetro é passado para o método, e com isso ele gera uma instrução SQL juntando os dados informados na primeira chamada através do método `set()` .

```
$sql = $this->builder->set('nome', 'Carlos Alberto')-
>getCompiledInsert('autores', FALSE);
// SQL: INSERT INTO autores (nome) VALUES ('Carlos Alberto')

$sql = $this->builder->set('bio', 'Desenvolvedor frontend')-
>getCompiledInsert();
// SQL: INSERT INTO autores (nome, bio) VALUES ('Carlos Alberto',
'Desenvolvedor frontend')
```

insertBatch()

Esse método é uma variação do `insert()` que permite a você executar a inserção de múltiplos registros no banco de dados através de uma única chamada de método.

```
$autores = [
    [
    'nome'   => 'Carlos Alberto',
```

```
    'cidade' => 'Vila Velha',
    'uf'     => 'ES',
    'bio'    => 'Desenvolvedor frontend'
    'status' => 1
    ],
    [
    'nome'   => 'Ana Cláudia',
    'cidade' => 'Vitória',
    'uf'     => 'ES',
    'bio'    => 'Desenvolvedora backend'
    'status' => 1
    ],
];
$this->builder->insertBatch($autores);
// SQL: INSERT INTO autores (nome, cidade, uf, bio, status) VALUES
('Carlos Alberto', 'Vila Velha', 'ES', 'Desenvolvedor frontend', 1), ('Ana
Cláudia', 'Vitória', 'ES', 'Desenvolvedora backend', 1)
```

Se você fosse utilizar o exemplo anterior com o método `insert()`, teria que executar um loop para pegar cada item do array `autores`.

```
$autores = [
    [
    'nome'   => 'Carlos Alberto',
    'cidade' => 'Vila Velha',
    'uf'     => 'ES',
    'bio'    => 'Desenvolvedor frontend'
    'status' => 1
    ],
    [
    'nome'   => 'Ana Cláudia',
    'cidade' => 'Vitória',
    'uf'     => 'ES',
    'bio'    => 'Desenvolvedora backend'
    'status' => 1
    ],
];

foreach($autores as $autor){
    $this->builder->insert($autor);
}
```

update()

Método utilizado para atualizar os dados de um determinado registro no banco de dados. Sua estrutura é similar à do método `insert()`, porém, na maioria das vezes ele trabalha em conjunto com o método `where()` de modo a determinar qual registro deve receber as alterações. Ele pode receber como parâmetro um array multidimensional ou um objeto.

```
$autor = [
    'bio'    => 'Gerente de TI'
    'status' => 0
];
$this->builder->where('id', 10);
$this->builder->update($autor);
// SQL: UPDATE autores SET bio = 'Gerente de TI', status = 0 WHERE id = 10
```

Há uma forma de utilizar o método `update()` sem chamar o método `where()`, que é passando a identificação do registro a ser alterado como segundo parâmetro, seja como uma string ou como um array.

```
$autor = [
    'bio'    => 'Gerente de TI'
    'status' => 0
];
$this->builder->update($autor, 'id = 10');
// SQL: UPDATE autores SET bio = 'Gerente de TI', status = 0 WHERE id = 10

// COMO ARRAY
$this->builder->update($autor, ['id' => 10]);
// SQL: UPDATE autores SET bio = 'Gerente de TI', status = 0 WHERE id = 10
```

getCompiledUpdate()

Método com funcionamento idêntico ao `getCompiledInsert()`. Em vez de executar a operação no banco de dados, ele retorna a string da instrução SQL correspondente à operação, que nesse caso é um `UPDATE`.

```
$autor = [
    'bio'    => 'Gerente de TI'
```

```
    'status' => 0
];
$this->builder->where('id', 10);
$this->builder->getCompiledUpdate($autor);
// SQL: UPDATE autores SET bio = 'Gerente de TI', status = 0 WHERE id = 10
```

updateBatch()

Esse método permite atualizar um conjunto de registros chamando uma única vez o método, da mesma forma como foi visto no método `insertBatch()`. Ele aceita tanto um array multidimensional quanto um objeto como primeiro parâmetro, e o segundo parâmetro é a coluna no banco de dados que representa a chave, ou seja, identifica o registro que deve ser alterado.

```
$autores = [
    [
    'id'     => 4,
    'status' => 0
    ],
    [
    'id'     => 6
    'status' => 1
    ],
];
$this->builder->updateBatch($autores, 'id');
// SQL: UPDATE autores SET status = CASE WHEN id = 4 THEN 0 WHEN id = 6
THEN 1 ELSE id END WHERE id IN (4, 6)
```

delete()

Esse método remove um registro do banco de dados e pode ser usado de forma combinada com o método `where()` ou então receber a identificação do registro a ser removido através do primeiro parâmetro.

```
$this->builder->where('id', 12);
$this->builder->delete();
// SQL :DELETE FROM autores WHERE id = 12
```

```php
$this->builder->delete(['id' => 5]);
// SQL :DELETE FROM autores WHERE id = 5
```

getCompiledDelete()

Método com funcionamento idêntico ao `getCompiledInsert()`, mas, em vez de executar a operação no banco de dados, ele retorna a string da instrução SQL correspondente à operação, que nesse caso é um `DELETE`.

```php
$this->builder->where('id', 12);
$this->builder->getCompiledDelete();
// SQL :DELETE FROM autores WHERE id = 12
```

replace()

Esse método é uma espécie de combinação dos métodos `delete()` e `insert()` executados um após o outro utilizando chaves primárias e únicas para determinar o registro a ser alterado.

Existem situações no desenvolvimento de aplicações web onde é preciso utilizar lógicas complexas combinando chamadas aos métodos `select()`, `update()`, `delete()` e `insert()` para que os dados sejam inseridos corretamente no banco de dados. Ao utilizar esse método, você elimina grande parte dessa lógica complexa e otimiza tanto o seu trabalho quanto o funcionamento do sistema.

```php
$autor = [
    'id'     => 12,
    'nome'   => 'Ana Cláudia',
    'cidade' => 'Vitória',
    'uf'     => 'ES',
    'bio'    => 'Desenvolvedora backend'
    'status' => 0
];
$this->builder->replace($autor);
// SQL: REPLACE INTO autores (nome, cidade, uf, bio, status) VALUES ('Ana
Cláudia', 'Vitória', 'ES', 'Desenvolvedora backend', 0)
```

emptyTable()

Método utilizado para remover os registros de uma tabela do banco de dados. Aceita como parâmetro o nome da tabela a ter os registros removidos, e caso você já tenha informado a tabela como foi feito neste capítulo, basta não informar o primeiro parâmetro.

```
$this->builder->emptyTable();
// SQL: DELETE FROM autores
```

truncate()

Remove todos os registros de uma tabela, assim como o método `emptyTable()`, mas além de limpar a tabela ele zera os contadores do tipo `AUTO INCREMENT`, coisa que o método `emptyTable()` não faz.

Aceita como parâmetro o nome da tabela a ser removida, e caso você já tenha informado a tabela como foi feito neste capítulo, basta não informar o primeiro parâmetro.

```
$this->builder->truncate();
// SQL: TRUNCATE autores
```

> **ATENÇÃO**
>
> Caso o comando `TRUNCATE` não esteja disponível no banco de dados que você está utilizando, será executado um `DELETE FROM tabela`.

Esse foi um longo capítulo, mas você conheceu todos os métodos do Query Builder disponíveis no CodeIgniter 4 e agora pode otimizar ainda mais o processo de desenvolvimento das suas aplicações.

CAPÍTULO 32
Transactions

Transactions, ou *transações*, é o nome dado a um conjunto de operações que compõem uma única tarefa a ser executada. No caso de bancos de dados, essas tarefas podem ser classificadas em 4 categorias: criação, leitura, atualização e exclusão, também conhecidas pela sigla CRUD (do inglês, *create*, *read*, *update* e *delete*). O uso de transactions faz com que o banco de dados garanta a execução correta de todas as tarefas e, caso alguma falhe, a transaction é abortada.

> **ATENÇÃO**
>
> A maioria das plataformas de banco de dados suporta transações de forma nativa, porém, se sua aplicação estiver utilizando MySQL você precisará utilizar os tipos de tabela InnoDB ou BDB em vez do mais comum, o MyISAM.

32.1 Como o CodeIgniter lida com as transactions

A forma como o CodeIgniter trabalha com as transactions é semelhante aos processos utilizados por uma popular classe de banco de dados chamada ADODB. Essa abordagem simplifica o processo de execução das transactions, permitindo que na maioria dos casos sejam necessárias apenas 2 linhas de código para tal operação.

Acompanhar as consultas e determinar se deve confirmar ou não uma transação a partir do sucesso ou falha delas é um grande

desafio e pode necessitar de uma boa quantidade de trabalho, principalmente se estiver lidando com consultas aninhadas.

Mas nessa versão do CodeIgniter foi implementado um sistema de transactions inteligentes que faz todo esse processo de forma automática (mas você também tem a opção de gerenciar manualmente).

Executando transactions

Para executar suas consultas utilizando transactions basta utilizar as funções `$this->db->transStart()` e `$this->db->transComplete()`. Você pode executar quantas consultas quiser entre as funções de inicialização e conclusão e todas passarão pelo processo de confirmação com base no sucesso da consulta.

```
$this->db->transStart();
$this->db->query('SELECT * FROM autores');
$this->db->query('SELECT * FROM posts');
$this->db->transComplete();
```

Por padrão, as transactions serão executadas em *Strict Mode*. Quando executadas nesse modo, se você estiver executando vários grupos de transactions e um grupo falhar, todos os grupos serão revertidos. Se o *Strict Mode* estiver desativado, esses grupos serão tratados de forma independente e assim a falha de um grupo não afetará a de outro.

Para desativar o *Strict Mode* basta adicionar a linha `$this->db->transStrict(false);` antes de chamar o método `$this->db->transStart()`.

Gerenciando erros

Caso a exibição de erros esteja ativada em `/app/Config/Database.php`, uma mensagem padrão de erro será exibida caso tenha ocorrido algum erro e a transaction não tenha sido confirmada. Se a exibição

de erros estiver desativada, então você poderá fazer o gerenciamento dos erros usando como base o exemplo a seguir:

```
$this->db->transStart();
$this->db->query('SELECT * FROM autores');
$this->db->query('SELECT * FROM posts');
$this->db->transComplete();

if ($this->db->transStatus() === FALSE)
{
        // aqui você pode criar um registro de log com informações
personalizadas
        log_message('error', 'Mensagem personalizada sobre o erro');
        // ou pode exibir o erro na tela através de alguma view criada
para essa finalidade
        return view('erros/transaction');
}
```

Desabilitando as transactions e usando o modo teste

As transactions já estão ativadas por padrão no CodeIgniter, mas se você precisar desativá-las basta adicionar a linha `$this->db->transOff()` antes de chamar o método `$this->db->transStart()`, dessa forma mesmo que haja uma instrução para utilizar transaction, ela será ignorada.

Com as transactions desativadas a confirmação das consultas acontece da mesma forma como se você estivesse executando no modo tradicional.

```
$this->db->transOff()

$this->db->transStart();
$this->db->query('SELECT * FROM autores');
$this->db->query('SELECT * FROM posts');
$this->db->transComplete();
```

Uma outra possibilidade que o CodeIgniter traz para o uso de transactions é o *Modo Teste*, onde as consultas são revertidas mesmo que haja confirmação para as consultas. Para fazer uso do

Modo Teste basta definir um parâmetro para o método `$this->db->transStart()` como `TRUE`.

```
$this->db->transStart(true);
$this->db->query('SELECT * FROM autores');
$this->db->query('SELECT * FROM posts');
$this->db->transComplete();
```

Executando as transactions manualmente

Você também pode executar as transactions manualmente, mas isso não trará nenhum benefício em relação à execução automática, muito pelo contrário, só vai fazer você escrever mais algumas linhas de código. Mas pode ser que em algum momento a aplicação que estiver desenvolvendo necessite dessa execução manual, e você pode aplicá-la conforme o exemplo a seguir:

```
$this->db->transBegin();

$this->db->query('SELECT * FROM autores');
$this->db->query('SELECT * FROM posts');

if ($this->db->transStatus() === FALSE)
{
    $this->db->transRollback();
}
else
{
    $this->db->transCommit();
}
```

Sempre que for executar manualmente uma transaction utilize o método `$this->db->transBegin()` no lugar do `$this->db->transStart()`.

Capítulo 33
Database Forge: Manipulando bancos de dados

Um dos recursos para trabalho com bancos de dados do CodeIgniter mais interessantes depois do Query Builder é a classe **Database Forge**. Ela possui métodos que permitem o gerenciamento da estrutura do banco de dados, criando, modificando e removendo tabelas, colunas e chaves estrangeiras.

Neste capítulo, você vai aprender como utilizar os métodos da classe Database Forge e tornar suas aplicações ainda mais robustas ou mesmo criar recursos que otimizem o seu processo de desenvolvimento de aplicações web.

33.1 Inicializando a classe

A classe Database Forge necessita do driver do banco de dados para funcionar, sendo assim ele já deve ter sido adicionado às configurações do autoload da aplicação.

Feito isso, você precisa apenas instanciar a classe `Forge` para ter acesso a todos os seus métodos. Por padrão, ela fará uso do banco de dados já definido nas configurações, mas caso você precise utilizá-la para se conectar a um banco de dados diferente do utilizado por padrão na aplicação, basta informar o nome de grupo do banco de dados, que deve estar previamente configurado em `/app/Config/Database.php` .

```
$forge = \Config\Database::forge();
//ou no caso de apontar um banco de dados diferente
$forge2 = \Config\Database::forge('identificacao_do_banco');
```

33.2 Criando e removendo bancos de dados

Para trabalhar diretamente com o banco de dados, a classe `Forge` possui 2 métodos: `createDatabase()` e `dropDatabase()`.

`$forge->createDatabase()`

Esse método cria um novo banco de dados e como primeiro parâmetro recebe o nome do banco a ser criado. Um segundo parâmetro pode ser adicionado (`true`) para adicionar uma verificação do tipo `IF EXISTS` ou fazer uma checagem de existência de algum banco de dados com o mesmo nome. A forma como o segundo parâmetro será tratado vai depender do SGBD (Sistema Gerenciador de Banco de Dados) utilizado.

```
$forge->createDatabase('nome_do_banco');
```

`$forge->dropDatabase()`

Esse método permite remover o banco de dados informado no primeiro parâmetro e todo o seu conteúdo.

```
$forge->dropDatabase('nome_do_banco');
```

Ambos os métodos retornam `true` ou `false`, permitindo que você crie regras de validação baseadas no retorno de suas execuções.

```
if ($forge->createDatabase('nome_do_banco'))
{
    // aqui entra sua rotina caso o banco tenha sido criado com sucesso
}

if ($forge->dropDatabase('nome_do_banco'))
{
    // aqui entra sua rotina caso o banco tenha sido removido com sucesso
}
```

33.3 Manipulando tabelas

Dentro das funcionalidades para criação de tabelas o CodeIgniter também disponibiliza métodos que permitem criar e remover colunas e chaves.

É importante se atentar a um detalhe dentro do fluxo de execução dos métodos, pois quando estamos operando manualmente dentro de um SGBD, geralmente temos uma tela única onde informamos o nome da tabela e suas respectivas colunas.

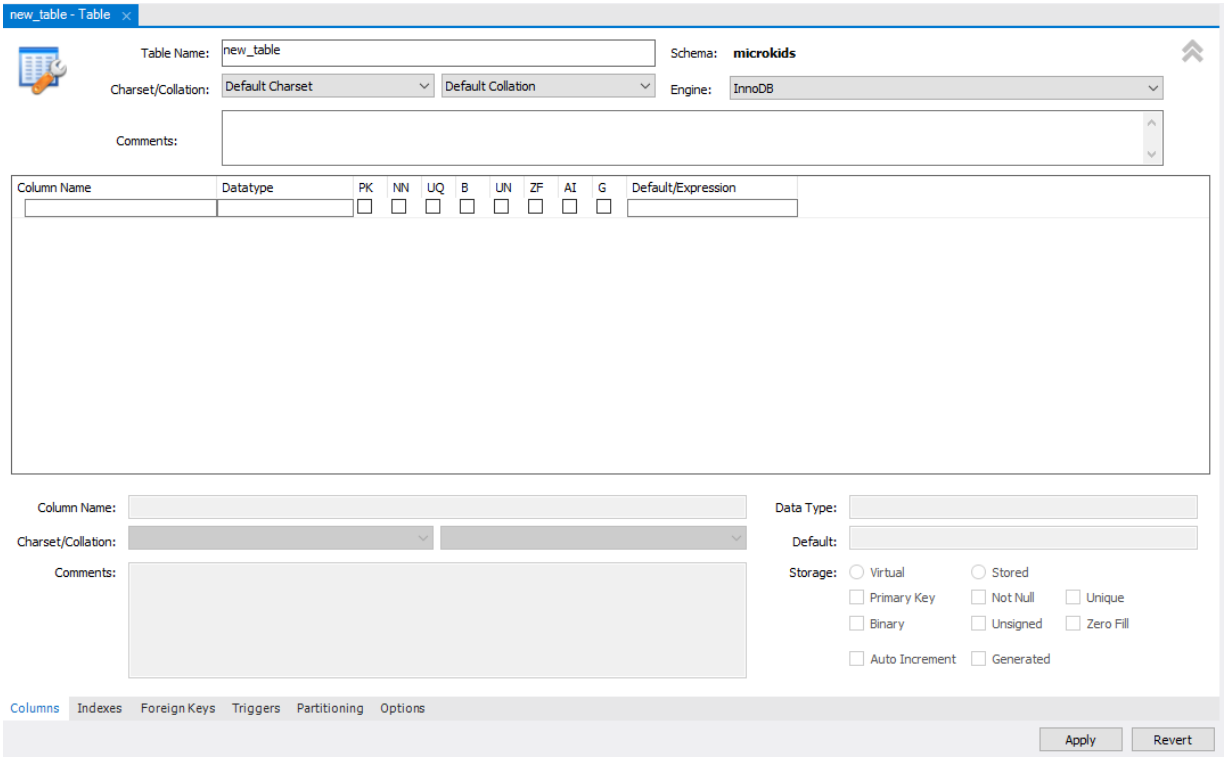


Figura 33.1: Tela para criação de tabela no MySQL Workbench 8.0 CE

Ao utilizar a classe `Forge` primeiro devemos criar as colunas e logo em seguida criar a tabela. Não se preocupe, o CodeIgniter possui um mecanismo dentro da classe para identificar as colunas criadas com o método `addField()` e associá-las automaticamente à tabela quando o método `createTable()` for chamado.

Criando colunas, chaves e tabelas

Os campos a serem criados devem ser estruturados em um array associativo contendo as informações sobre cada coluna, como `type`, `constraint`, `auto_increment` e outros.

```
$fields = [
    'id'          => [
        'type'           => 'INT',
        'constraint'     => 5,
        'unsigned'       => true,
        'auto_increment' => true
    ],
    'titulo'       => [
        'type'           => 'VARCHAR',
        'constraint'     => '255',
        'unique'         => true,
    ],
    'subtitulo'      => [
        'type'           =>'VARCHAR',
        'constraint'     => 160,
        'default'        => true,
    ],
    'descricao' => [
        'type'           => 'TEXT',
        'null'           => true,
    ],
    'status'      => [
        'type'           => 'ENUM',
        'constraint'     => ['publicado', 'pendente', 'rascunho'],
        'default'        => 'rascunho',
    ],
];
```

Veja a seguir uma lista com as informações aceitas pela classe `Forge` na criação de colunas:

- `type` : tipo de dado que a coluna poderá receber, por exemplo, `INT`, `VARCHAR`, `TEXT`, `DOUBLE`, `ENUM`;
- `unsigned` : `true` para definir a coluna como "UNSIGNED";
- `default` : valor padrão da coluna quando nenhum for informado na inserção de conteúdo;

- `null` : `true` para o caso de a coluna não precisar de dados e `false` caso ela tenha o preenchimento obrigatório;
- `auto_increment` : `true` para adicionar valores incrementais à coluna quando o seu tipo permitir, como `INT` , por exemplo;
- `unique` : `true` para gerar uma chave exclusiva para a coluna.

Com os campos definidos, é hora de executar o método `addField()` acompanhado do método `createTable()` para criar a estrutura da tabela.

```
$forge->addField($fields);
$forge->createTable('nome_da_tabela');
```

O método `createTable` aceita até 3 parâmetros, sendo o primeiro o nome da tabela, o segundo (`true` ou `false`) para indicar se deverá ou não ser feita uma verificação de existência de tabela com o mesmo nome, e o terceiro, um array associativo contendo os atributos da tabela, por exemplo a `engine` .

```
$forge->createTable('nome_da_tabela', TRUE, ['ENGINE' => 'InnoDB']);
```

> **ATENÇÃO**
>
> No caso do MySQL, se você não especificar os atributos `CHARACTER SET` e/ou `COLLATE` para o método `createTable()` , serão sempre utilizados os valores padrões configurados no MySQL.

Ainda falando sobre a criação de colunas, além do método `addField()` que já foi visto, a classe `Forge` possui mais 4 outros métodos muito úteis: `addKey()` , `addPrimaryKey()` , `addUniqueKey()` e `addForeignKey()` .

`addKey()`

Esse método é utilizado para adicionar uma ou mais chaves à tabela, e como primeiro parâmetro recebe o nome da coluna que deve ser definida como chave. Já o segundo parâmetro, opcional, pode ser definido como `true` para tornar a chave como primária, e o

terceiro parâmetro, se definido como `true`, tornará a chave exclusiva. Caso você precise de uma chave primária com várias colunas, então o primeiro parâmetro deve ser um array contendo o nome das colunas a serem definidas como chave primária.

```
$forge->addKey('id_autor', TRUE);
```

addPrimaryKey()

Esse método é utilizado para criar uma chave primária na tabela e recebe como parâmetro o nome da coluna que deve ser definida como chave primária.

```
$forge->addPrimaryKey('id');
```

addUniqueKey()

Esse método é utilizado para criar uma chave única na tabela e recebe como parâmetro o nome da coluna que deve ser definida como chave única. Caso seja necessário passar mais de uma coluna como parâmetro, então elas devem ser inseridas em um array, e esse array passado como parâmetro.

```
$forge->addUniqueKey('titulo');
$forge->addUniqueKey(['titulo', 'subtitulo']);
```

addForeignKey()

Esse método é o responsável por inserir chaves estrangeiras na tabela. Esse tipo de chave ajuda a definir o relacionamento entre as tabelas e até mesmo ações. Ele aceita 3 parâmetros: o nome da coluna a ser definida como chave estrangeira; o nome da tabela de origem da chave; e a coluna na tabela de origem que estará associada à coluna a ser criada.

```
$forge->addForeignKey('id_autor', 'autores', 'id');
```

Você também pode especificar a ação desejada para as propriedades da restrição, por exemplo, quando for executado um `update` ou `delete`.

```
$forge->addForeignKey('id_autor', 'autores', 'id', 'CASCADE', 'CASCADE');
```

ATENÇÃO

Os 4 métodos que acabaram de ser apresentados, sempre que forem chamados na aplicação devem ser precedidos de uma chamada ao método `createTable()`. Uma forma bastante útil de utilizar esse conjunto de métodos para criar suas tabelas é seguindo a sequência adiante, adicionando os métodos conforme a necessidade:

1. criar o array com as colunas
2. adicionar a(s) chave(s) com o método `addKey()`
3. (opcional) adicionar outras informações de chave através dos métodos `addPrimaryKey()`, `addUniqueKey()` e `addForeignKey()`
4. Criar a tabela com o método `createTable()`

Se por acaso você precisar remover uma chave estrangeira da tabela, basta executar o método `dropForeignkey()`, passando como primeiro parâmetro o nome da tabela e como segundo parâmetro o nome da coluna que representa a chave estrangeira a ser removida.

```
$forge->dropForeignKey('posts','id_autor');
```

Removendo e renomeando tabelas

Para remover uma tabela basta chamar o método `dropTable()` e informar como primeiro parâmetro o nome da tabela e no segundo parâmetro (opcional) `true` para o caso de desejar verificar se a tabela realmente existe no banco de dados antes de executar a instrução de remoção.

```
$forge->dropTable('posts', TRUE);
```

Já para renomear uma tabela, basta chamar o método `renameTable()` e passar para ele como primeiro parâmetro o nome atual da tabela e

como segundo, o novo nome da tabela.

```
$forge->renameTable('nome_atual', 'novo_nome');
```

Adicionando colunas em uma tabela

Para adicionar colunas em uma tabela já criada anteriormente você deverá utilizar o método `addColumn()`. Ele recebe como primeiro parâmetro o nome da tabela onde a coluna deve ser adicionada, e como segundo, um array multidimensional igual ao que foi utilizado para criar as colunas da tabela, informando a coluna a ser criada.

```
$fields = [
    'descricao_seo' => [
        'type' => 'TEXT'
    ]
];
$forge->addColumn('posts', $fields);
```

Se você estiver utilizando o MySQL, então tem a vantagem de poder definir a posição do campo a ser criado em relação a outro, passando como uma das chaves do array o atributo `after` ou o atributo `first` para adicionar a coluna na primeira posição dentro da estrutura da tabela.

```
$fields = [
    'descricao_seo' => [
        'type'  => 'TEXT',
        'after' => 'descricao'
    ],
    'tags' => [
        'type'  => 'TEXT',
        'first' => true
    ]
];
$forge->addColumn('posts', $fields);
```

Removendo colunas em uma tabela

Para remover colunas em uma tabela, basta utilizar o método `dropColumn()` e informar como primeiro parâmetro o nome da tabela e no segundo, o nome da coluna a ser removida.

```
$forge->dropColumn('posts', 'tags');
```

Modificando colunas em uma tabela

`modifyColumn()`

Esse método funciona de forma idêntica ao `addColumn()` inclusive no que diz respeito aos parâmetros que recebe, a diferença é que, em vez de criar uma nova coluna, ele vai modificar as colunas informadas no array multidimensional. Caso você queira alterar o nome da coluna, basta informar o atributo `name` no array.

```
$fields = [
    'titulo'       => [
        'type'           => 'VARCHAR',
        'constraint'     => 100,
    ],
    'subtitulo'      => [
        'type'           =>'VARCHAR',
        'constraint'     => 160,
        'default'        => true,
    ],
    'descricao' => [
        'type'           => 'LONGTEXT',
        'name'           => 'descricao_seo'
    ]
];

$forge->modifyColumn('posts', $fields);
```

Capítulo 34
Migrations: Mantendo o banco de dados estruturado e organizado

Quando você está trabalhando em um projeto junto de uma equipe, é muito comum utilizar o versionamento de código para manter todos os envolvidos com os arquivos atualizados e poder gerenciar as mudanças, erros, testes, deploy, enfim, cuidar bem do processo de desenvolvimento.

Porém, fazer o controle de versão do banco de dados é um grande desafio, pois muitas vezes alguém pode precisar inserir, modificar ou remover alguma tabela ou coluna e, se fizer isso diretamente na estrutura do banco de dados, pode acabar esquecendo de informar aos demais envolvidos de que foi feita uma mudança estrutural no banco de dados.

Para ajudar a evitar esse tipo de problema e manter o banco de dados atualizado e as alterações devidamente documentadas e organizadas é que o CodeIgniter possui o recurso de **Migrations & Seeds** que você verá neste capítulo.

34.1 Migrations

As migrations são uma forma de mapear as alterações executadas mantendo a estrutura do banco de dados devidamente atualizada para todos os envolvidos, pois ao configurar a execução das migrations o próprio framework vai cuidar de verificar quando foi a última alteração e se existe alguma nova a ser aplicada.

Configurando a execução das migrations

No CodeIgniter 4, por padrão, as migrations já vêm ativadas, mas caso queira deixar esse recurso desabilitado basta você acessar o arquivo de configuração em `app/Config/Migrations` e informar o valor `false` para a variável `$enabled`.

Ainda no arquivo de configuração, você pode alterar o nome da tabela padrão onde ficarão armazenadas as informações sobre a execução das migrations, permitindo que o framework gerencie de forma eficiente os upgrades e downgrades da estrutura. A definição da tabela é feita na variável `$table`. Uma outra configuração disponível e importante, principalmente porque ela influencia na definição do nome do arquivo físico da migration, é a forma como será construído o prefixo de cada uma. Na variável `$timestampFormat` você poderá optar por 3 formas de definir o modo como o prefixo será escrito.

```
YmdHis_
Y-m-d-His_
Y_m_d_His_
```

Criando uma migration

As migrations são executadas em ordem numérica de acordo com o método definido, podendo seguir para a frente (crescente) ou para trás (decrescente). Cada migration é criada e identificada a partir de uma marcação de data e hora correspondente ao exato momento em que foi definida, seguindo o formato `AAAAMMDDHHIISS`, por exemplo, `20191216074835`. Essa é uma forma de evitar conflitos de numeração quando se está trabalhando em equipe.

Essa marcação de data e hora funciona como um prefixo para as migrations que devem ter seu nome complementado por um underline (_) logo após o prefixo e uma descrição relacionada à alteração que essa migration fará no banco de dados, por exemplo, `20191216074835_criacao_tabela_comentarios.php`.

O prefixo também pode ter suas partes separadas por traço (-) ou mesmo por um underline (_).

```
20191216074835_criacao_tabela_comentarios.php
2019-12-16-07-48-35_criacao_tabela_comentarios.php
2019_12_16_07_48_35_criacao_tabela_comentarios.php
```

Todas as migrations devem ser armazenadas no diretório `app/Database/Migrations/` e ser uma classe estendida de `\CodeIgniter\Database\Migration`. Além disso, é necessário que possuam 2 métodos, o `up()`, que aplica a alteração à estrutura do banco de dados, e o `down()`, que volta o banco de dados para a estrutura anterior à execução da migration.

Dentro da classe criada com a estrutura da migration, você terá acesso aos métodos do Query Builder e da classe `Forge`, respectivamente `$this->db` e `$this->forge`, podendo fazer uso dos recursos disponíveis no framework para execução de instruções SQL e manipulação de banco de dados.

```php
<?php namespace App\Database\Migrations;

class AddBlog extends \CodeIgniter\Database\Migration {

    protected $DBGroup = 'nome_do_grupo';

    public function up()
    {
        $this->forge->addField([
                'id'          => [
                        'type'           => 'INT',
                        'constraint'     => 5,
                        'unsigned'       => TRUE,
                        'auto_increment' => TRUE
                ],
                'post_id'          => [
                        'type'           => 'INT',
                        'constraint'     => 5,
                ],
                'usuario_id'          => [
                        'type'           => 'INT',
                        'constraint'     => 5,
                ],
```

```
                'comentario'        => [
                        'type'           => 'VARCHAR',
                        'constraint'     => '100',
                ]
        ]);
        $this->forge->addKey('id', TRUE);
        $this->forge->createTable('comentarios');
    }

    public function down()
    {
        $this->forge->dropTable('comentarios');
    }
}
```

As migrations são sempre executadas no banco de dados padrão da aplicação, mas caso as alterações definidas na migration sejam para um banco de dados específico cujo grupo já tenha sido definido nas configurações, basta você especificar através da variável de classe `$DBGRoup` o nome do grupo, que o CodeIgniter se encarregará de aplicar as mudanças de forma exclusiva nele.

```
<?php namespace App\Database\Migrations;

class AddBlog extends \CodeIgniter\Database\Migration {

    protected $DBGroup = 'nome_do_grupo';

    public function up()
    {
        // código para atualização
    }

    public function down()
    {
        // código para reversão
    }
}
```

Lidando com chaves estrangeiras em uma migration

Umas das coisas que mais geram problemas no uso de migrations são as chaves estrangeiras, pois elas podem gerar erros ao tentar eliminar tabelas ou colunas da estrutura do banco de dados. Mas no CodeIgniter você pode eliminar temporariamente as verificações de chave estrangeira durante a execução das migrations através dos métodos `disableForeignkeyCheck()` e `enableForeignKeyCheck()`.

```php
<?php namespace App\Database\Migrations;

class AddBlog extends \CodeIgniter\Database\Migration {

    protected $DBGroup = 'nome_do_grupo';

    public function up()
    {
        $this->db->disableForeignKeyCheck();

        $this->forge->addField([
                'id'          => [
                        'type'           => 'INT',
                        'constraint'     => 5,
                        'unsigned'       => TRUE,
                        'auto_increment' => TRUE
                ],
                'post_id'          => [
                        'type'           => 'INT',
                        'constraint'     => 5,
                ],
                'usuario_id'          => [
                        'type'           => 'INT',
                        'constraint'     => 5,
                ],
                'comentario'       => [
                        'type'           => 'VARCHAR',
                        'constraint'     => '100',
                ]
        ]);
        $this->forge->addKey('id', TRUE);
        $this->forge->createTable('comentarios');
```

```
        $this->db->enableForeignKeyCheck();
    }

    public function down()
    {
        $this->forge->dropTable('comentarios');
    }
}
```

Migrations na linha de comando

Muitos desenvolvedores preferem trabalhar utilizando a linha de comando sempre que possível, e o CodeIgniter permite o trabalho com as migrations a partir da linha de comando. É um recurso opcional, pois você pode continuar criando as migrations de forma manual como foi mostrado anteriormente, mas esse recurso pode facilitar bastante o processo.

A seguir você verá os principais métodos utilizados para trabalhar com migrations na linha de comando.

`migrate`

Executa as migrations disponíveis atualizando o banco de dados padrão configurado na aplicação. Na mesma linha de comando você pode definir algumas opções:

- `-g` : para definir o grupo do banco de dados onde a migration deve ser executada;
- `-n` para definir o namespace, (diferente de `App`) que é o padrão;
- `-all` : para executar a migration para todos os namespaces definidos.

```
php spark migrate -g db_test -n Blog
```

Ao utilizar a opção `-all` , todos os namespaces serão verificados em busca de migrations que ainda não foram executadas, sendo coletados e classificados como um grupo ordenado por data de

criação, reduzindo possíveis conflitos entre a aplicação principal e módulos da aplicação.

`rollback`

Faz com que a estrutura do banco de dados retorne para a forma anterior à última migration executada. Nessa linha de comando você pode utilizar 2 opções:

- `-g` : para definir o grupo do banco de dados onde a migration deve ser executada;
- `-b` : para escolher o batch, ou seja, o número do batch para retorno que pode ser positivo ou negativo.

```
spark migrate:roolback -b 1
```

`refresh`

Aplica uma atualização no banco de dados revertendo todas as migrations e depois reexecutando cada uma delas. Assim como `migrate` , aceita 3 opções de configuração na linha de comando:

- `-g` : para definir o grupo do banco de dados onde a migration deve ser executada;
- `-n` para definir o namespace, (diferente de `App`) que é o padrão;
- `-all` : para executar a migration para todos os namespaces definidos.

```
php spark migrate:refresh
```

`status`

Exibe uma lista com todas as migrations e a data e hora de execução, ou então um `-` se não foi executada. Você pode utilizar a opção `-g` com essa linha de comando para especificar o grupo de banco de dados a ser verificado.

```
php spark migrate:status
Filename                                              Migrated On
20191216074835_criacao_tabela_comentarios.php   2019-12-16 08:34:47
```

create

Cria automaticamente o esqueleto do arquivo da migration em `/app/Database/Migrations` adicionando a data e hora como prefixo ao nome do arquivo. Essa linha de comando aceita a opção `-n` para definir o namespace, assim como em outras linhas de comando já apresentadas.

```
php spark migrate:create nome_do_arquivo
```

34.2 Seeds

Os seeds nada mais são do que uma forma de inserir registros no banco de dados durante o processo de desenvolvimento sem a necessidade de utilizar um SGBD e ainda mantendo um histórico como acontece com as migrations. Eles ainda permitem que você utilize dados estáticos sem ter que incluí-los nas migrations, como tabelas com dados geográficos (estados, cidades, países), informações sobre eventos, e muitos outros que você pode identificar conforme a necessidade das aplicações que estiver desenvolvendo.

Programaticamente falando, seeds são classes simples estendidas de `CodeIgniterDatabaseSeeder` e compostas por um método `run()`. Dentro desse método você pode criar formas de dados de acordo com a necessidade da aplicação, uma vez que a classe tem acesso aos recursos do Query Builder através de `$this->db` e aos recursos da classe `Forge` através de `$this->forge`.

Os arquivos de seeds são armazenados em `/app/Database/Seeds` e o nome do arquivo deve corresponder ao nome da classe. No caso do exemplo a seguir, o nome do arquivo deve ser `PrimeiroSeeder.php`.

```
<?php namespace App\Database\Seeds;

class PrimeiroSeeder extends \CodeIgniter\Database\Seeder
```

```
{
    public function run()
    {
        $data = [
                'nome'  => 'Jonathan Lamim',
                'email' => 'contato@jonathanlamim.com.br'
        ];

        // Instrução SQL simples
        $this->db->query("INSERT INTO autores (nome, email)
VALUES(:username:, :email:)", $data);

        // Usando o Query Builder
        $this->db->table('autores')->insert($data);
    }
}
```

É possível chamar um seed dentro de outro através do método `call()`, permitindo que você organize seus seeds a partir de um seed central, por exemplo.

```
<?php namespace App\Database\Seeds;

class SegundoSeeder extends \CodeIgniter\Database\Seeder
{
        public function run()
        {
                $this->call('PrimeiroSeeder');
        }
}
```

Também é possível utilizar um nome de classe para informar qual seed será executado através do método `call()`, recurso muito útil quando se está trabalhando com uma aplicação modularizada.

```
$this->call('nome-da-classe\Database\Seeds\nome-do-seed');
```

Para executar o seed, você utilizará a linha de comando de forma muito parecida com a utilizada para as migrations.

```
php spark db:seed SegundoSeeder
```

Feito isso, o banco de dados será alimentado com as informações definidas no seed.

Estendendo o framework

Estamos chegando ao final deste livro e agora que você já conhece o CodeIgniter 4 de forma suficiente para criar aplicações de qualidade com segurança e agilidade, vou lhe mostrar como estender o framework para que além dos recursos nativos dele você possa utilizar seus próprios recursos como se fizessem parte do *Core* do CodeIgniter.

Capítulo 35
Estendendo classes do Core

Sempre que o CodeIgniter é executado na sua aplicação, um conjunto de classes é inicializado de forma automática como parte da estrutura principal da aplicação.

Muitas vezes é necessário que durante essa inicialização sejam carregadas classes e funcionalidades que vão além da estrutura principal, que são uma extensão do que o framework já traz como base. Através da extensão de classes do framework você pode implementar as funcionalidades específicas da aplicação sem perder o que já é parte do padrão principal.

Mas é preciso atenção na hora de estender classes que compõem a base do framework, pois, em caso de falhas de codificação, o carregamento da estrutura principal do framework será afetado.

O CodeIgniter 4 traz as seguintes classes como parte da estrutura principal:

- `Config\Services`
- `CodeIgniter\Autoloader\Autoloader`
- `CodeIgniter\Config\DotEnv`
- `CodeIgniter\Controller`
- `CodeIgniter\Debug\Exceptions`
- `CodeIgniter\Debug\Timer`

- `CodeIgniter\Events\Events`
- `CodeIgniter\HTTP\CLIRequest` (se for chamado através da linha de comando)
- `CodeIgniter\HTTP\IncomigRequest` (se for chamado através de requisição HTTP)
- `CodeIgniter\HTTP\Request`
- `CodeIgniter\HTTP\Response`
- `CodeIgniter\HTTP\Message`
- `CodeIgniter\HTTP\URI`
- `CodeIgniter\Log\Logger`
- `CodeIgniter\Log\Handlers\BaseHandler`
- `CodeIgniter\Log\Handlers\FileHandler`
- `CodeIgniter\Router\RouteCollection`
- `CodeIgniter\Router\Router`
- `CodeIgniter\Security\Security`
- `CodeIgniter\View\View`
- `CodeIgniter\View\Escaper`

35.1 Substituindo classes do Core

Para substituir o uso de uma classe padrão do framework por sua própria classe, você deve se certificar dos seguintes pontos:

- o `Autoloader` é capaz de encontrar a sua classe;
- sua classe estende a interface apropriada;
- o `Service` apropriado foi modificado para carregar a sua classe no lugar da classe principal.

Vamos supor que na sua aplicação você queira utilizar a sua própria classe `app\Libraries\RouteCollection`. Para começar, ela deverá ter uma estrutura de código base conforme a apresentada a seguir, com `namespace` e definição da classe de interface, além da implementação do método para carregamento das rotas:

```php
<?php namespace App\Libraries;

use CodeIgniter\Router\RouteCollectionInterface;

class RouteCollection implements RouteCollectionInterface
{
    public static function routes(bool $getShared = true)
    {
        if ($getShared)
        {
            return static::getSharedInstance('routes');
        }

        return new RouteCollection(static::locator(), config('Modules'));
    }
}
```

35.2 Estendendo classes do Core

Se você precisa que sua aplicação tenha recursos adicionais aos já existentes em alguma, ou algumas classes do framework, o que você deve fazer é estender a classe em vez de criar uma nova classe com todos os métodos já existentes e substituir o carregamento.

Quando você estende uma classe, está fazendo quase a mesma coisa que a substituição, exceto pelo fato de que *a nova classe tem a obrigatoriedade de estender a classe pai*.

Vamos seguir utilizando o exemplo da classe `RouteCollection`, mas agora, em vez de usar uma própria, vamos estender a nativa. A declaração dela deve ser a seguinte:

```php
<?php namespace App\Libraries;

use CodeIgniter\Router\RouteCollection;
```

```
class RouteCollection extends RouteCollection
{

}
```

Repare que:

- o `namespace` foi mantido.
- em vez de utilizar `use CodeIgniter\Router\RouteCollectionInterface;` foi utilizado `use CodeIgniter\Router\RouteCollection;`
- em vez de declarar `class RouteCollection implements RouteCollectionInterface` foi declarado `class RouteCollection extends RouteCollection`

Se for necessário o uso de um método construtor `__construct()` na sua extensão da classe, lembre-se de estendê-lo do método construtor da classe pai.

```
<?php namespace App\Libraries;

use CodeIgniter\Router\RouteCollection;

class RouteCollection extends RouteCollection
{
    public function __construct()
    {
        parent::__construct();
    }
}
```

Ao estender uma classe, se algum dos métodos dessa classe estendida tiver o nome idêntico ao da classe pai, ele substituirá o método principal, então ao criar sua classe estendida certifique-se de que não tem métodos nomeados de forma idêntica, exceto se a intenção for de fato substituir o método principal.

Capítulo 36
Estendendo o controlador

O controlador principal do CodeIgniter não deve ser modificado em hipótese alguma, mas ao iniciar uma aplicação, você já terá à disposição uma extensão desse controlador chamada `app/Controllers/BaseController.php`.

É a partir desse controlador que todos os demais controladores da sua aplicação deverão ser estendidos, de modo a aproveitar os componentes pré-carregados e qualquer funcionalidade adicional que você fornecer.

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Home extends BaseController {

}
```

Você pode ter outros controladores estendidos do controlador principal `Controller`, e isso não tem problema nenhum, desde que a declaração seja feita da forma correta.

```
<?php namespace App\Controllers;

use CodeIgniter\Controller;

class Admin extends Controller {

}
```

Ao estender um controlador você também pode fazer o pré-carregamento de componentes como helpers, models, bibliotecas, serviços entre outros.

Para definir o que será pré-carregado junto com o controlador, basta informar para as variáveis relacionadas a cada tipo de componente a ser carregado.

Se você quer carregar helpers, vai chamar:

```
protected $helpers = ['html', 'text', 'form'];
```

Se você quer carregar bibliotecas, vai chamar:

```
protected $libraries = ['email'];
```

Quaisquer outros componentes a serem carregados ou mesmo dados a serem processados devem ser adicionados ao construtor `initController()`.

```
public function initController(\CodeIgniter\HTTP\RequestInterface
$request, \CodeIgniter\HTTP\ResponseInterface $response,
\Psr\Log\LoggerInterface $logger)
{
    parent::initController($request, $response, $logger);
    $this->session = \Config\Services::session();
}
```

Dessa forma você pode construir seus próprios controladores de base sem perder as funcionalidades nativas dos controladores no CodeIgniter.

Capítulo 37
Eventos

O recurso de eventos do CodeIgniter 4 é uma evolução dos hooks utilizados no CodeIgniter 3. Eles são um meio de explorar e modificar o funcionamento interno da estrutura sem a necessidade de alterar os arquivos principais.

Toda vez que uma aplicação com CodeIgniter é executada, ela segue um processo específico, e pode haver casos em que seja necessária a execução de alguma operação em determinado estágio desse processo de carregamento.

É através dos eventos que essas operações a serem executadas durante o carregamento da aplicação são definidas. Eles funcionam com um padrão de publicação/assinatura onde o evento é acionado em algum momento específico durante a execução da aplicação. Para definir o que deverá ser executado e em que evento essa execução ocorrerá você deve registrar as operações na classe `Events`.

37.1 Definindo um evento

O arquivo `app/Config/Events.php` é onde são definidas as ações a serem executadas. Para isso é utilizado o método `on()` passando como parâmetro o nome do evento e o que deverá ser executado quando o evento em questão for disparado.

Veja a seguir um exemplo de como fazer a configuração de uma ação relacionada ao evento `pre_system`, que ocorre antes de o sistema ser carregado.

```
Events::on('pre_system', ['SuaClasse', 'SuaFuncao']);
```

O método `on()` aceita também um terceiro parâmetro, que é responsável por definir a prioridade de execução, já que é possível executar múltiplas ações em um mesmo evento. O valor passado como terceiro parâmetro corresponde à ordem de execução e deve ser um inteiro. Quanto menor o valor, maior a prioridade, sendo que o menor valor aceito é 1.

```
Events::on('pre_system', ['SuaClasse', 'SuaFuncao', 10]);
```

Caso existam assinaturas de execução com o mesmo valor para o terceiro parâmetro, será considerada a ordem em que foram definidas em `app/Config/Events.php` .

Também é possível fazer uso de 3 constantes predefinidas no framework para definir a prioridade:

```
define('EVENT_PRIORITY_LOW', 200);
define('EVENT_PRIORITY_NORMAL', 100);
define('EVENT_PRIORITY_HIGH', 10);
```

Durante a execução dos eventos e suas respectivas assinaturas, se um valor `false` for retornado, a execução será interrompida.

***Event points* disponíveis:**

- `pre_system` : chamado logo no início da execução da aplicação e somente as classes `benchmark` e `Events` foram carregadas até então, nenhum roteamento ou outro processo ocorreu.
- `post_controller_constructor` : chamado imediatamente após a instanciação do controlador, antes de qualquer outra chamada acontecer.
- `post_system` : chamado logo depois que a página renderizada é enviada ao browser, quando não há nada mais a ser processado do lado do servidor.

37.2 Criando seus próprios eventos

Através da classe `Events` é possível criar eventos personalizados através do método `trigger()`. Você pode definir qualquer quantidade de parâmetros para as assinaturas e os assinantes receberão esses parâmetros na mesma ordem em que foram definidos.

```
\CodeIgniter\Events\Events::trigger('evento_personalizado', $foo, $bar,
$baz);

Events::on('evento_personalizado', function($foo, $bar, $baz) {
        ...
});
```

O CodeIgniter permite que você faça simulação de eventos durante os testes de sua aplicação, evitando que rotinas longas e demoradas sejam executadas enquanto testes são realizados.

Através do método `simulate()`, você pode dizer ao framework que deseja apenas a simulação dos eventos e não a execução real deles, bastando passar como parâmetro para `simulate()` valores como `TRUE` ou `FALSE`.

Capítulo 38
Criando bibliotecas

Como já foi visto anteriormente, o CodeIgniter traz cerca de 15 bibliotecas nativas para auxiliar no processo de desenvolvimento das aplicações. Mas elas nem sempre são suficientes para atender às necessidades do projeto e então novas bibliotecas precisam ser criadas.

No CodeIgniter 4 é muito simples criar bibliotecas, e é isso que você vai aprender neste capítulo.

38.1 A estrutura das bibliotecas

Bibliotecas nada mais são de que uma coleção de métodos organizados em uma classe, capazes de executar rotinas específicas dentro da aplicação.

As bibliotecas que você criar para sua aplicação devem ser armazenadas em `/app/Libraries` e ter uma estrutura base como a demonstrada a seguir:

```
<?php namespace App\Libraries;

class NomeDaClasse
{

}
```

Ela segue a estrutura de uma classe, com namespace que pode ser o padrão utilizado pelas bibliotecas do CodeIgniter 4 (`App\Libraries`) ou então um outro que você prefira.

As bibliotecas também podem ser extensões de bibliotecas padrões do CodeIgniter 4, como foi visto no capítulo *Estendendo classes do Core*.

38.2 Criando uma biblioteca

Para exemplificar esse processo vamos criar uma classe que vai exibir na tela banners em diversos tamanhos e cores, tendo como base os exemplos de banners disponibilizados no site (https://www.lipsum.com/banners/).

> Foi feito o download dos arquivos GIF de cada um dos banners, conforme eles disponibilizam para que todos os arquivos estivessem no ambiente local.

Crie um arquivo chamado `Banners.php` em `/app/Libraries` e em seguida adicione o código a seguir a ele. Esse código é a estrutura base para a criação de bibliotecas no CodeIgniter 4 com o método que retornará o link do banner a ser exibido a partir dos parâmetros `$cor` e `$tamanho`. As imagens dos banners foram armazenadas em `/public/banners`.

```
<?php

namespace App\Libraries;

class Banners
{
    public function getBanner(string $cor, string $tamanho)
    {
        $banner = $cor."_".$tamanho.".gif";

        return base_url("banners/$banner");
```

```
    }
}
```

Logo após criar o arquivo e inserir o código nele, a biblioteca já está disponível para uso, bastando fazer a chamada dela no código, como no exemplo a seguir:

```
$libBanner = new Banners;
$urlBanner = $libBanner->getBanner('black', '970x90');
echo "<img src='$urlBanner'/>";
```

38.3 Estendendo uma biblioteca

Uma classe estendida é basicamente igual à classe que você acabou de criar, porém ela precisa ter a informação de qual classe ela estende:

```
<?php namespace App\Libraries;

use \Namespace\Da\Classe\De\Extensao

class NomeDaClasse extends NomeDaClasseDeExtensao
{

}
```

O restante do processo de extensão de uma biblioteca é o mesmo da criação de uma nova biblioteca, criando os devidos métodos. Caso queira substituir algum método da biblioteca que está estendendo, basta criar um novo método com o mesmo nome. A classe estendida também fica armazenada em `/app/Libraries`.

Veja a seguir uma estrutura para uma classe que é uma extensão da classe `Time`:

```
<?php
```

```php
namespace App\Libraries;

use \CodeIgniter\I18n\Time;

class MyTimes extends Time
{

}
```

O ponto mais importante para a criação de bibliotecas no CodeIgniter 4 é compreender sua estrutura e fazer uso correto dos namespaces, o restante nada mais é do que código PHP e funcionalidades nativas do próprio framework.

Capítulo 39
Criando helpers

Os *Helpers* são conjuntos de funções para otimizar o processo de escrita de código, tornando-o mais fluído e enxuto. E o processo de criação ou extensão de helpers no CodeIgniter é bem simples. A diferença entre estender um helper e criá-lo como um novo está no nome do arquivo.

Quando quiser estender um helper a partir de um já existente no CodeIgniter basta utilizar o mesmo nome de arquivo de um helper já existente. Independente de criar um novo ou estender de um já existente, os helpers que você criar devem ficar armazenados em `/app/Helpers` .

39.1 Estendendo um helper

No código a seguir estamos criando uma extensão para o helper `number` nativo do CodeIgniter onde adicionaremos duas outras funções para identificar se um número é par ou ímpar. São funções bem simples, mas a intenção é mostrar como criar e estender helpers para que você possa utilizar o recurso de acordo com sua necessidade.

Crie um novo arquivo em `/app/Helpers` com o nome de `number_helper.php` e adicione a ele o código a seguir:

```
<?php

// verifica se é par
if (!function_exists('number_is_even')) {
    function number_is_even($num)
    {
        return ($num % 2 == 0) ? true : false;
```

```
    }
}

// verifica se é ímpar
if (!function_exists('number_is_odd')) {
    function number_is_odd($num)
    {
        return ($num % 2 == 0) ? false : true;
    }
}
```

Feito isso, para utilizar essas novas funcionalidades adicionadas ao helper `Number`, basta você carregá-lo para a aplicação com `helper('number')` e depois chamar os métodos `number_is_even()` e `number_is_odd()` conforme a necessidade.

39.2 Criando um helper

O procedimento para criar um helper único, que não se estende de nenhum existente no CodeIgniter é o mesmo, você apenas deve nomear o arquivo com um nome que não é utilizado por nenhum helper nativo.

Crie um novo arquivo em `/app/Helpers` com o nome `arroba_helper.php` e adicione o código a seguir, onde temos uma função que substituirá todas as letras "a" minúsculas de um texto por um arroba (@).

Poderíamos fazer isso estendendo o helper `Text` já que se trata de uma função relacionada a textos, mas a intenção aqui é apenas mostrar como criar um helper único.

```
<?php

if (!function_exists('a_to_arroba')) {
    function a_to_arroba($text)
    {
        return str_replace("a", "@", $text);
```

```
    }
}
```

Para utilizar as funcionalidades do helper que acabou de criar, basta você carregá-lo para a aplicação com `helper('arroba')` e depois chamar os métodos `a_to_arroba()` conforme a necessidade.

Lista dos nomes de helpers nativos até a versão 4.0.2

- `array_helper.php`
- `cookie_helper.php`
- `date_helper.php`
- `filesystem_helper.php`
- `form_helper.php`
- `html_helper.php`
- `inflector_helper.php`
- `number_helper.php`
- `security_helper.php`
- `text_helper.php`
- `url_helper.php`
- `xml_helper.php`

Você pode ver o código-fonte desses exemplos de helpers e seu uso fazendo o download dos arquivos no repositório oficial no GitHub. https://github.com/jlamim/livro-codeigniter4

Capítulo 40
Trabalhando com arquivos de tradução

O CodeIgniter oferece um recurso nativo que permite você desenvolver aplicações multi-idioma sem complicações, e o principal recurso para fazer isso é o dos **arquivos de tradução**.

Através desses arquivos você pode estruturar todo o conteúdo de texto da sua aplicação, incluindo mensagens de erro para validação de dados, em arquivos separados organizados em diretórios de acordo com o idioma.

40.1 Configurando o uso de traduções

Para que os arquivos de tradução sejam lidos e utilizados pela aplicação é preciso realizar algumas configurações no sistema.

No arquivo `/app/Config/App.php` você terá a variável `$defaultLocale`, ela é responsável por informar ao sistema qual é o idioma padrão, ou seja, que arquivos de tradução devem ser utilizados pela aplicação logo que ela for carregada.

Você também pode fazer uma configuração para que a aplicação gerencie de forma automática o conteúdo definindo duas configurações adicionais em `/app/Config/App.php`.

Na variável `$negotiateLocale` você deve definir o valor `true` para que a classe `Request` possa fazer o gerenciamento das traduções.

Agora altera a variável `$supportedLocales` passando os idiomas suportados pela aplicação.

```
$supportedLocales = ['en', 'es', 'pt-BR'];
```

Após essa ativação a aplicação passará a gerenciar de forma automática as traduções de acordo com as que foram informadas em `$supportedLocales` e caso alguma tradução não seja encontrada, será utilizada a informação a partir do dicionário do primeiro idioma informado em `$supportedLocales`.

Definição do idioma através da URL

Muitas aplicações utilizam um nó na URL para informar o idioma, como por exemplo `seusite.com/en` ou `seusite.com/es`.

Para que isso funcione na aplicação, basta utilizar a identificação `{locale}` na hora de definir as rotas da aplicação.

```
$routes->get('{locale}/', 'Home::index');
```

Dessa forma o CodeIgniter entenderá que a URL base da aplicação é composta pelo própria URL base definida nas configurações da aplicação mais o idioma, de acordo com o que foi definido em `$supportedLocales`, mantendo assim a integridade das rotas e sem a necessidade de ficar fazendo filtros dentro do controlador.

E se uma rota for acessada sem a informação do idioma, então será utilizado o idioma padrão definido em `$defaultLocale`.

Recomendações

É recomendado utilizar os códigos de idioma BCP 47, de forma a seguir um padrão de nomenclatura. No capítulo **Links úteis** tem um link para o site do W3C com mais informações sobre BCP 47.

Geralmente, usam-se nomenclaturas como "es", "en", "pt" mas há casos em que existe uma variação do idioma, e aí a nomenclatura ficaria "en-US", "en-GR", "pt-BR" ou outras variações do mesmo idioma para países diferentes.

O CodeIgniter tem um sistema de seleção de idioma inteligente que direciona o carregamento das traduções fazendo com que traduções

para "en-US" ou "en-GR" sejam carregadas do diretório `en` caso não exista um diretório com a nomenclatura idêntica à do idioma.

Então você não precisa obrigatoriamente ter arquivos duplicados caso vá utilizar uma única tradução para o inglês por exemplo, só por conta do padrão de nomenclatura.

A estrutura de um arquivo de tradução

Os arquivos de idioma ficam armazenados em `/app/Language` e dentro desse diretório devem ser inseridos os diretórios de cada idioma da aplicação e então no diretório do idioma são criados os arquivos de tradução.

```
/app
    /Language
        /pt-BR
            app.php
        /en
            app.php
        /es
            app.php
```

Um arquivo de tradução no CodeIgniter 4 nada mais é do que um array contendo como chave a referência que será utilizada para obter a tradução, e o valor sendo o texto a ser exibido.

```
<?php
return [
    'headTitulo'         => 'Exemplos de Código - CodeIgniter 4',
    'headDescricao'      => 'Exemplos de código para apoiar o conteúdo do
livro CodeIgniter 4',
    'conteudoTitulo'     => 'Multi-idioma',
    'conteudoDescricao'  => 'Nesse exemplo você verá como construir uma
estrutura de conteúdo multi-idioma usando o CodeIgniter 4 e seus recursos
nativos.',
    'conteudoLinkPt'     => 'Português',
    'conteudoLinkEn'     => 'Inglês',
    'conteudoLinkEs'     => 'Espanhol',
    'footerComprarLivro' => 'Comprar Livro',
```

```
    'footerSiteAutor'    => 'Site do Autor',
];
```

Você pode ter diversos arquivos de tradução, então explore essa possibilidade e organize as traduções para que fique mais fácil de gerenciá-las conforme a aplicação for crescendo.

> Se sua aplicação tiver outros idiomas além do português, lembre-se sempre de ter os mesmos arquivos de tradução em todos os diretórios de idioma, caso contrário, a aplicação apresentará erro.

40.2 Utilizando os arquivos de tradução

Após criar os arquivos, você precisará de uma simples chamada para exibir a tradução: `lang()` .

Através da chamada `lang()` na view você exibirá o texto de acordo com o idioma selecionado. Essa função recebe como parâmetro a identificação do termo a ser traduzido utilizando a sintaxe de pontos.

```
lang('App.headTitulo')
```

No exemplo anterior, temos `App.headTitulo` , onde `App` é o nome do arquivo de tradução para o idioma selecionado e `headTitulo` é a chave no array com as traduções.

Na maioria dos casos a sintaxe será essa, mas pode acontecer de você precisar utilizar um array dentro de outro para uma melhor organização, principalmente para arquivos de tradução de mensagens de erro. Sendo assim, podemos ter algo como:

```
// Arquivo: Validation.php
<?php
return [
```

```
    'nome' => [
        'required'   => 'Campo de preenchimento obrigatório.',
        'min_length' => 'O nome precisa ter pelo menos 3 caracteres'
    ];
```

E para chamar a tradução a partir do dicionário você pode utilizar o seguinte código dentro do seu model:

```
class UsuarioModel extends Model
{
    protected $validationRules    = [
        'nome' => 'required| min_length[15]'
    ];

    protected $validationMessages = [
        'nome'    => [
            'required'   => 'Validation.nome.required',
            'min_length' => 'Validation.nome.min_length'
        ]
    ];
}
```

40.3 Passando parâmetros para o dicionário

Em algumas situações é necessário enviar parâmetros para compor as mensagens de tradução e torná-las compreensíveis. No CodeIgniter 4 é possível fazer isso sem mudar o jeito de criar os arquivos de tradução e chamar a tradução na view.

No arquivo de tradução você precisará informar no texto a ser exibido o ponto onde esses parâmetros entram e qual o tipo de dado usando a sintaxe `{posicao_no_array, tipo_de_dado}` .

```
<?php
// Arquivo: Messages.php
return [
    'cadastroSucesso'           => 'O usuário {0, string} foi cadastrado
```

```
com sucesso.',
    'registrosExcluidosSucesso' => '{0, number} registro(s) excluído(s)
com sucesso.',
    'saldoAtualizado'           => '{1, number} ponto(s) adicionado(s)
para o usuário {0, string}.'
]
```

Na hora de chamar o método `lang()`, basta passar como segundo parâmetro um array contendo os valores a serem substituídos no dicionário.

```
lang('Messages.cadastroSucesso', ['Jonathan Lamim']);
lang('Messages.registrosExcluidosSucesso', [3]);
lang('Messages.saldoAtualizado', ['Jonathan Lamim', 350]);
```

Repare que no arquivo de tradução para `saldoAtualizado` utilizamos 2 parâmetros e eles estão fora da ordem passada no array quando a função `lang('Messages.saldoAtualizado', ['Jonathan Lamim', 350])` foi chamada.

Não é obrigatório você estruturar de forma sequencial no dicionário, mas é preciso respeitar a posição ao chamar a função `lang()`, caso contrário, o dicionário vai exibir os dados de forma incorreta.

Você também pode usar uma outra forma, que é até mais segura, para recuperar os parâmetros no array e substituí-los no dicionário. Para isso, em vez de usar a sintaxe `{posicao_no_array, tipo_de_dado}` utilize o `{chave_no_array, tipo_de_dado}`, assim, independente da ordem em que os dados forem organizados no array, o dicionário vai utilizar a chave para obter a informação a ser adicionada ao texto.

```
<?php
// Arquivo: Messages.php
return [
    'cadastroSucesso'           => 'O usuário {nomeUsuario, string} foi
cadastrado com sucesso.',
    'registrosExcluidosSucesso' => '{numRegistros, number} registro(s)
excluído(s) com sucesso.',
    'saldoAtualizado'           => '{totalPontos, number} ponto(s)
```

```
adicionado(s) para o usuário {nomeUsuario, string}.'
]

lang('Messages.cadastroSucesso', ['nomeUsuario' => 'Jonathan Lamim']);
lang('Messages.registrosExcluidosSucesso', ['numRegistros' => 3]);
lang('Messages.saldoAtualizado', ['nomeUsuario' => 'Jonathan Lamim',
'totalPontos' => 350]);
```

Você pode utilizar os seguintes dados para substituição dentro do dicionário:

- **numbers:** integer, currency, percent
- **dates:** short, medium, long, full
- **time:** short, medium, long, full
- **spellout:** número como string (ex. 34 se torna trinta e quatro)
- **ordinal:** número ordinal
- **duration:** duração, tempo

> Esses tipos de dados estão de acordo com a documentação oficial do ICU para a biblioteca utilizada.

Definindo um idioma específico para uso em uma tradução

Pode parecer estranho ter que definir um idioma para uma tradução quando a aplicação já faz isso de forma automática. Mas imagine um cenário onde sua aplicação, independente do idioma em que o usuário está visualizando os dados na tela, precise exibir o preço de um produto usando uma notação específica.

A aplicação é uma loja virtual para uma empresa de importação onde o idioma pode ser alterado mas a moeda utilizada é sempre o dólar. Você pode resolver esse problema padronizando a notação na hora de chamar a função `lang()` na view passando um terceiro parâmetro que representa o idioma da notação da moeda.

```
echo lang('{price, number, currency}', ['price' => 9.99], 'en-US');
// Resultado: $9.99
```

40.4 Como utilizar as traduções padrões do CodeIgniter

Para facilitar o trabalho, o CodeIgniter conta com arquivos de traduções das mensagens padrões para 19 idiomas (até o momento em que este livro foi finalizado) e que são mantidas tanto pela equipe de desenvolvimento quanto pela comunidade. Você tem duas formas de instalar esses arquivos de tradução em sua aplicação: por download manual ou via Composer.

Instalando via download

Para instalar via download basta você acessar https://github.com/codeigniter4/translations e fazer o download do arquivo compactado com todas as traduções. Após concluir o download, descompacte o arquivo e copie o diretório `Language` para dentro de `/app` caso queira todas as traduções disponíveis, ou então copie apenas os diretórios referentes aos idiomas desejados para `/app/Language`. Feito isso você terá disponíveis as traduções e basta aplicar as configurações de acordo com o que foi explicado no início deste capítulo.

Instalando via Composer

Para instalar via Composer basta acessar o diretório raiz da sua aplicação no terminal e então executar o comando `composer require codeigniter4/translations`. Nesse caso serão copiadas todas as traduções disponíveis para dentro da sua aplicação, mas não no diretório `/app/Language` e sim como um pacote do próprio CodeIgniter.

Esse tipo de instalação não muda em nada a forma como você aplica as configurações e faz uso dos dicionários, ele apenas mantém um padrão na organização e agiliza o processo.

Capítulo 41
Links úteis

- Site oficial do CodeIgniter: https://codeigniter.com
- Fórum oficial do CodeIgniter: https://forum.codeigniter.com/
- Documentação oficial do CodeIgniter: https://codeigniter.com/user_guide/index.html
- Repositório do CodeIgniter 4 no GitHub: https://github.com/codeigniter4
- Site oficial do Composer: https://getcomposer.org/
- Site oficial do PHP-FIG: https://php-fig.org

Sites com tutoriais de CodeIgniter

- https://tutoriaiscodeigniter.com.br (em português)
- https://dev.to/t/codeigniter (em inglês)

Outros links úteis

- https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status
- http://regular-expressions.info
- https://www.w3.org/International/articles/language-tags/
- https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classMessageFormat.html#details

41.1 Exemplos de código disponíveis

Para ajudar você a desenvolver ainda mais os conhecimentos e habilidades relacionados ao CodeIgniter 4 estou disponibilizando no repositório oficial do livro no GitHub uma série de exemplos de código. Para acessá-los basta ir até lá e fazer o download dos arquivos ou então um *fork* do repositório.

https://github.com/jlamim/livro-codeigniter4

Veja a seguir uma lista com os exemplos disponíveis para ajudar você em seus estudos.

- Criação de controllers
- Criação de bibliotecas e helpers
- Validação de formulários
- Upload de arquivos
- Manipulação de imagens
- Envio de email
- Requisições com cURL nativo
- Multi-idioma
- CRUD
- Limitação de requisições (throttler)

Capítulo 42
Conclusão

Chegamos ao final deste livro e eu espero que você já esteja desenvolvendo aplicações de forma produtiva e ágil com o CodeIgniter 4. O que foi visto aqui é o conhecimento essencial e fundamental para que você possa obter o máximo de proveito e performance dessa nova versão do CodeIgniter, e em hipótese alguma é tudo o que se deve saber.

Eu mesmo, a todo instante, estou aprendendo algo novo sobre esse framework que tem feito parte da minha stack de desenvolvimento há muitos anos. Ficarei muito grato se você puder me enviar um depoimento sobre este livro, isso vai me ajudar a saber se o objetivo o livro está sendo alcançado. Para o enviar o depoimento basta acessar o link abaixo:

https://livrocodeigniter.com.br/depoimentos

Que todo o conhecimento adquirido durante essa leitura possa ampliar suas oportunidades profissionais e potencializar sua carreira.

Sucesso!!!